USENIX

NASHVILLE CONFERENCE PROCEEDINGS

SUMMER

1991

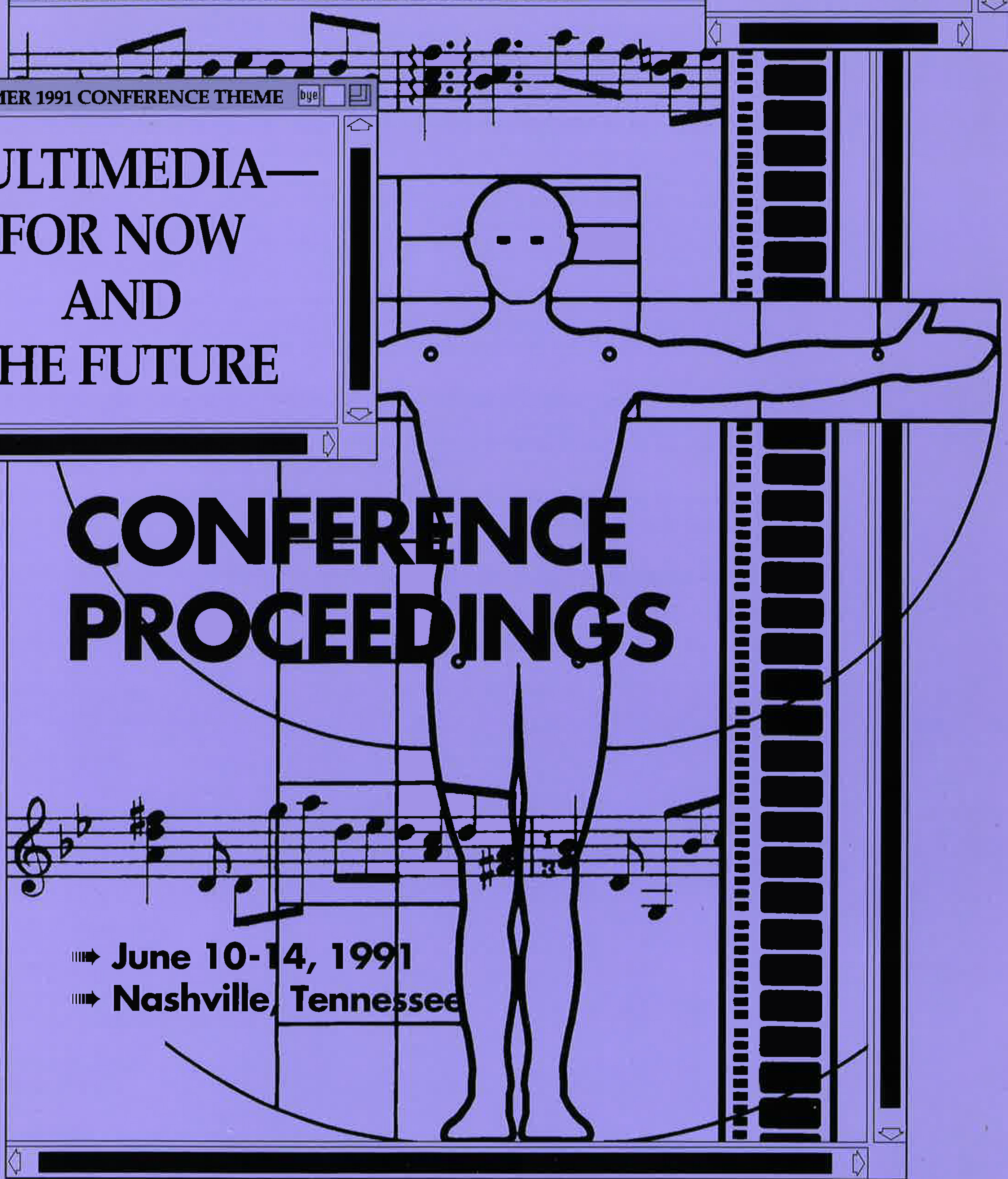
USENIX

SUMMER 1991 TECHNICAL CONFERENCE & EXHIBITION

SUMMER 1991 CONFERENCE THEME

MULTIMEDIA— FOR NOW AND THE FUTURE

CONFERENCE PROCEEDINGS



➠ June 10-14, 1991

➠ Nashville, Tennessee

Proceedings of the Summer 1991 USENIX Conference

USENIX ASSOCIATION

June 10-14, 1991
Nashville, Tennessee, U.S.A.

For additional copies of these proceedings write:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is $32 for members and $38 for nonmembers.
Outisde the U.S. A. and Canada, please add
$22 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

1991 Winter Dallas
1990 Summer Anaheim
1990 Winter Washington, DC
1989 Summer Baltimore
1989 Winter San Diego
1988 Summer San Francisco
1988 Winter Dallas
1987 Summer Phoenix
1987 Winter Washington, DC
1986 Summer Atlanta
1986 Winter Denver
1985 Summer Portland
1985 Winter Dallas
1984 Summer Salt Lake City
1984 Winter Washington, DC
1983 Summer Toronto
1983 Winter San Diego

TABLE OF CONTENTS

MULTIMEDIA DEMOS

Chair: Jun Murai
Keio University

MULTIMEDIA PUBLISHING I

Chair: Mike Hawley
MIT Media Lab

MULTIMEDIA DATA RATES AND SYNCHRONIZATION

Chair: Charles Roberts
Hewlett-Packard

PANEL

MULTIMEDIA DEMO

Chair: Lisa Bloch
Sun User Group

CONFERENCE COMMITTEE

TECHNICAL PROGRAM COMMITTEE

Deborah K. Scherrer, *Technical Program Chair*
mt Xinu

Eric P. Allman, *UC Berkeley*
Frances Brazier, *Vrije Universiteit*
Tom Duff, *AT&T Bell Laboratories*
Daniel E. Geer, *Digital Equipment Corporation*
Stanley P. Hanks, *Technology Transfer Associates*
Michael Hawley, *MIT Media Lab*
Jun Murai, *Keio University*
Sharon Murrell, *AT&T Bell Laboratories*
Alan G. Nemeth, *Digital Equipment Corporation*
Jeff Peck, *Sun Microsystems, Inc.*
Charles E. Perkins, *IBM T. J. Watson Research Center*
Gretchen Phillips, *SUNY – Buffalo*
Charles S. Roberts, *Hewlett-Packard*
Larry Stead, *Bellcore*
Avadis Tevanian, *NeXT, Inc.*

PROCEEDINGS PRODUCTION

Carolyn S. Carr, *Proceedings Coordination*
USENIX Association
Jaap Akkerhuis, *Typesetting*
mt Xinu

TECHNICAL PROGRAM REVIEWERS

Susan Angebranndt, *Digital Equipment Corporation*
Paul DeBra, *University of Eindhoven, Holland*
Stu Feldman, *Bellcore*
Bob Gray, *US West*
Blaine Garst, *NeXT Inc.*
Chet Jusczak, *Digital Equipment Corporation*
Jackie Kasputys, *Digital Equipment Corporation*
Steve Law, *Digital Equipment Corporation*
Andy Litman, *NeXT Inc.*
Alan Marcum, *NeXT Inc.*
Rick McGowan, *NeXT Inc.*
M. Kirk McKusick, *UC Berkeley*
Dave Moore, *NeXT Inc.*
John Morse, *Digital Equipment Corporation*
Brian Pinkerton, *NeXT Inc.*
Larry Palmer, *Digital Equipment Corporation*
John Puttress, *AT&T Bell Laboratories*
Chris Saether, *Digital Equipment Corporation*
Jim McGinness, *Digital Equipment Corporation*
Jon Reeves, *Digital Equipment Corporation*
Bob Travis, *Digital Equipment Corporation*
Leslie Wharton, *Digital Equipment Corporation*
Gayn Winters, *Digital Equipment Corporation*

CONFERENCE ORGANIZER

Judith F. DesHarnais, *Meeting Planner*
USENIX Association

PREFACE

We wanted Nashville to be special, unique, perhaps even wildly exciting. We chose our theme "Multimedia — For Now and the Future" because the challenges of incorporating and integrating voice, video, animated graphics, touch, and music into a single environment are now upon us. These new media add immeasurably to the power of the computer for expression of ideas, while the new interfaces multiply system complexity and enlarge the challenge of providing easy access to computer resources.

We wanted to explore the questions:

- How can system designers and multimedia developers work together to deliver, support, and fully integrate the new media?
- What are the technical engineering requirements of enabling UNIX and advanced operating systems to process effectively the new types of data?
- How can we design new multimedia interfaces to improve information handling?
- How can we make these resources and media more accessible to the user?

Such a program required a different approach. We had to be pro-active, locating and working closely with the best multimedia researchers to bring their knowledge and experiences to the operating systems community.

I was very selective in choosing my program committee, and each member brings a special talent or expertize. These people were sent out to bring in the best and brightest of the multimedia and systems researchers. Of the over 100 submissions we received, close to half were solicited, coaxed, or cajoled from their authors in some way. From the large body of submissions, we were equally selective, and chose only 28.

A new addition to the USENIX program is the multimedia demonstrations. We wanted to find and demonstrate projects that offered insight into the directions being taken in developing fully-integrated multimedia systems. We worked with both vendors and multimedia research labs to put together live presentations of their most interesting projects. We offered them extended timeslots in the program, relaxed the usual "new, unpublished research" requirements, and helped them secure and arrange the sometimes exotic audio and visual requirements. Demonstrators were given the option of submitting a full paper for the proceedings. Some projects lent themselves to this, others did not, but you will find here at least an abstract or brief description of these presentations, and in several cases full papers.

Another change to the format was the inclusion of the "Concurrent Session" talks into a fully-integrated program. As these presentations are generally mini-tutorials or interactive work-sessions, they are outside the scope of the peer-review process and thus are not included in the proceedings. Special thanks go to Sharon Murrel and Andrew Hume for working graciously with us to develop this coordinated program.

We chose one panel for inclusion in the refereed program: addressing the timely, and emotionally-charged, issues of intellectual property. You'll find a brief description in these proceedings.

Please do take note of the individuals on the program committee — each of these people made grand and glorious contributions of time, energy, and unbounded enthusiasm for this project. Special thanks also go to Alan Nemeth for suggesting the theme. And I would like to commend Sam Leffler personally for patiently bringing me up to speed on the state of the multimedia community. Demonstrators of the multimedia projects deserve special laudits for the immense amount of time spent preparing their presentations and transporting their equipment from far away places to show us their projects. Of course, none of this would be possible without the wonderfully supportive and tireless efforts of the USENIX staff, including Judy DesHarnais our conference organizer, newcomer Cynthia Deno, who designed and produced our creative new literature, and Carolyn Carr and Jaap Akkerhuis for their painstaking production of the proceedings. Finally, my friends at mt Xinu should be highly commended for graciously and generously allowing me the time and resources needed to pull a conference of this nature together. Thank you one and all.

We hope you enjoy the program.

Deborah K. Scherrer
Program Chair

AUTHOR INDEX

KEYNOTE: Musical Dreams and Musical Reality

Paul Lansky
Princeton University
Music Department
Princeton, N.J. 08544-1007
paul@winnie.princeton.edu

By now it is common knowledge that computers are wonderful music machines. At this point it probably true that most of the music we hear either has some computer mediation (as in digital recording), or is actually being created with the help of one cpu or another (particularly in popular music). But what has been emerging in the minds of denizens of loftier domains (composers, for example) is that we are beginning to see ways of reconstructing our fundamental views of what music is all about as a result of capabilities which are only now beginning to come into our hands. This talk will attempt to elucidate some of these new concepts with examples and descriptions. Hardware and software requirements for the music of the future will also be briefly considered.

Paul Lansky has emerged as one of the most prominent figures in the Computer Music field, being a pioneer in digital sound synthesis techniques, particularly the technique known as linear prediction (LPC). His articles, editorial contributions and book reviews in *Perspectives of New Music* have brought him recognition as a theorist, author and critic as well as a composer. Lansky also performed French Horn in the Dorian Woodwind Quintet between 1965 and 1967. Recently, Lansky has received recognition for the computer software programs (CMIX) he initially developed for his own work.

Long-Term Caching Strategies for Very Large Distributed File Systems

Matt Blaze
Rafael Alonso

Princeton University
Department of Computer Science
mab@princeton.edu

ABSTRACT

This paper examines the feasibility of using long term (disk based) caches in very large distributed file systems (DFSs). We begin with an analysis of file access patterns in a distributed Unix workstation environment, and identify properties of use to the DFS designer. We then introduce long-term caching strategies that maintain consistency while dramatically reducing the load on file servers. We describe a number of algorithms for maintaining client caches, and present the results of a trace-driven simulation that shows how relatively small disk-based caches can be used to reduce server traffic by 60% to 90%. Finally, we outline possible mechanisms for dynamically organizing these caches into adaptive hierarchies to allow arbitrary scaling of the number of clients and the use of low-bandwidth communication networks. A small (2 or 3 level) hierarchy, coupled with smart caching techniques, has the potential to reduce traffic by an order of magnitude or more over a flat scheme.

1. Introduction and Motivation

Distributed File Systems (DFSs), such as Sun NFS [3], have gained wide acceptance as a mechanism for sharing files among small groups of computers. Although the advantages of a shared file system structure are well established, DFSs are seldom used with very large numbers of machines or over low-bandwidth, heterogeneous, unreliable, or geographically distant networks. While the advantages of a DFS could apply equally well to such large scale systems, the assumptions made by current DFS software do not "scale up". In particular, current systems require file server help for most client reads, so both the server and the network must be able to handle a load proportional to the amount of client activity. In order for a DFS to scale, most client activity must be hidden from the server. We seek a system in which, for example, all users (in the world) of a particular version of Unix could mount a single centrally maintained /bin directory, or in which the remote mount replaces mail and ftp as the method of choice for sharing files among machines spread across great distances.

In first-generation DFSs (such as NFS), the only time the file server does not have to intervene for a client request is when the data being read is already in the client cache. Because the caches are small (and fit in memory), only very recently accessed files and directory information are usually cached. Furthermore, cached data is flushed after a very short time (typically about 30 seconds) to minimize consistency problems. Although these caches do have an impact on client performance, they are not sufficiently large or persistent to enable servers to handle arbitrarily large numbers of clients.

A number of experimental file systems [2][8][9] aim at scalability, but no one system attempts to scale to arbitrarily large numbers of autonomous, heterogeneous clients on arbitrary networks.

1.1. Caching vs. Replication

Two methods of distributing data to client machines without requiring the server to serve every request are replication and caching. In the traditional sense, replication involves propagation of files to the remote sites as they change. When a file changes, the new data (or at least a message invalidating the old data) must be propagated to each replica. A replication strategy may be static, where the files (or file systems) to be replicated and the machines on which the copies are to be stored are determined in advance. This results in potentially large amounts of redundant traffic when frequently written-to files are propagated to clients who may never read them. Static replication strategies require careful planning and monitoring if they are to be used to maximum advantage. The administrative overhead of determining what files to replicate, plus the cost of replicating inappropriate files, makes it suitable mainly for widely used files known not to change very often (such as /vmunix). Several experimental systems make use of static replication.

Caching, on the other hand, can be viewed as a dynamic replication scheme ordinarily done to improve client performance by keeping copies of the most recently used data in memory. This helps only with files opened in the very recent past. Because of the short-term nature of most file system caches, consistency and update propagation are typically managed by simply limiting the maximum age of a file in the cache to a very short time.

Both static replication and caching can improve client performance, but for the purposes of designing a highly scalable system, this is not the right metric at which to look. A better measure is to look at how much load each client puts on its file server. A system can be said to be scalable only if the server can handle a very large number of clients without itself becoming a bottleneck. In a small DFS, it is reasonable (both in terms of performance and economy of hardware) for most requests to be served remotely; the success of NFS proves this. However, as systems grow beyond a few dozen machines on a local network, it becomes attractive to have local disk copies of the remote data. We seek a middle ground between static replication of entire file systems and short term memory-based caches.

2. Distributed File Access Patterns

Virtually all efforts to build scalable DFSs include some kind of replication or caching scheme. Caching helps to reduce traffic (or server load) only if files are accessed in a fairly consistent manner. This section attempts to identify the kinds of access patterns present in current distributed systems. Knowledge of such patterns is essential for designing and evaluating a potential cache algorithm.

The success of a caching strategy depends upon two factors: the ability to predict when a file will be next used, and the ability to predict when the data will become invalid by being overwritten or deleted. Fortunately, Unix files accesses tend to be surprisingly predictable in practice, and this regularity can be exploited in designing a DFS. Past studies [4][5][7] have found that Unix files are normally quite small (1k-10k) and that most accesses are of the entire file. This suggests that caching entire files is reasonable. It is also known that many files are changed or deleted within a few minutes of creation, which suggests that a replication scheme that blindly copies new files to all potential clients is not the best strategy.

Although the results of past analysis of Unix file access patterns has been fairly consistent, less is known about the behavior of distributed file systems where the same files are available for use by many processors. We analyzed a week-long trace of file accesses conducted at DEC/SRC and kindly made available to us. This was augmented by a small trace conducted locally. The data consisted of a log of all system calls issued by about 120 Unix workstations in a computing research environment. We did not look at individual read and write calls; instead, to simplify our analysis, we looked only at open and unlink system calls. We consider an open for read system call to be a non-destructive "read" operation and an open for write or open for read/write or unlink system call to be a destructive "write" operation.

We identify three properties of file access from this trace data that, if generalized, make very efficient caching strategies feasible. The first property, locality, is analogous to the well-known property of memory-reference patterns, and says simply that files tend to be accessed from the same places from which that have been recently opened. The second, which we call "inertia", says that files tend to be accessed in the same manner as previous accesses. The third, "entropy", says that files become less "volatile" over time and tend to become read-only as they are read more often. We discuss these properties in detail

below.

2.1. "Locality"

Intuition suggests that the most likely machine to open a file is one that has accessed it recently. We find this property to be very strong in the DEC trace data; about 90 percent of reads are on workstations that have read the file in the past.

Most reads and writes are of files that are used at only a few workstations, even including "system" files like /bin/sh and /vmunix. This is particularly true of writes.

Figure 1 plots opens for writing (and unlink) against the number of machines that have read the file since it was last written (or unlinked). Note the sharp decline in writes after just 2 machines have read the file.

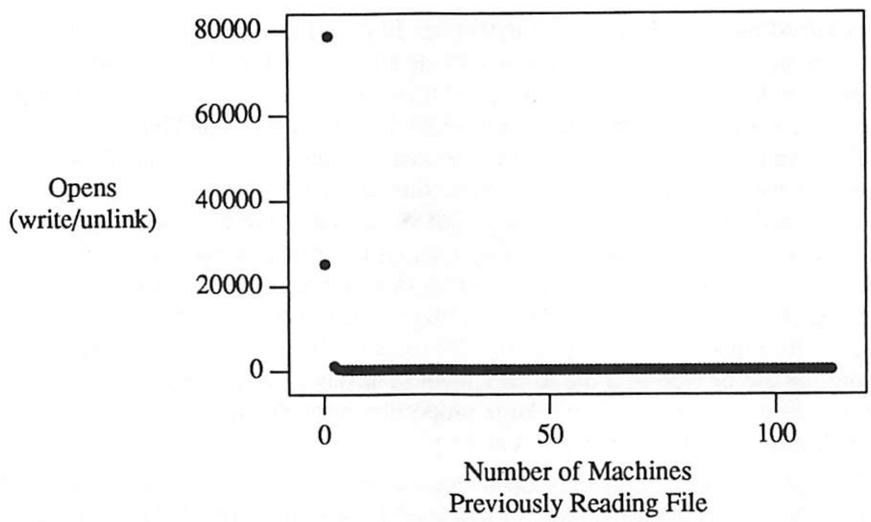


Figure 1 - Writes vs. Number of Machines

Figure 2 plots opens for read only against the number of machines that have previously opened the file since it was last written.

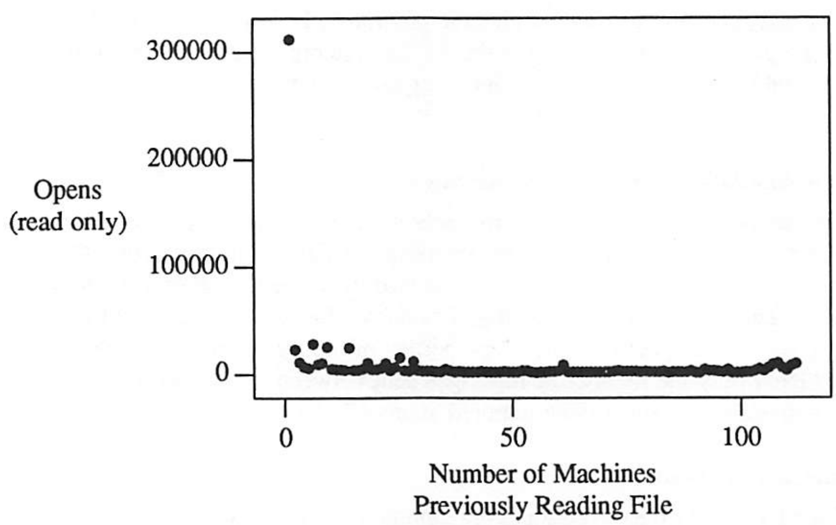


Figure 2 - Reads vs. Number of Machines

Locality is also quite strong in the sense that the previous machine to access a file is also very likely to be the next machine to do so. Almost 70% (429758 out of 623383) of file accesses are by the same machine that last opened the file.

2.2. "File Inertia"

Files in the trace had a strong tendency to be opened in the same mode as they were opened last, particularly for reading. Of 582302 read operations, 519204, or 89%, followed a read operation on that file. For writing, of the 104629 writes logged, 77267, or 73%, overwrote files that had never been read since they were last written. That is, 73% of write operations wrote data that were never read during the trace period. (Part of this is accounted for by writes to append-only log files and opens of write only devices such as /dev/tty).

2.3. "File Entropy"

As files are read more (and to a lesser extent, as they age) they become less likely to be written or deleted before they are read again. Put another way, as a file is read more, it becomes increasingly likely that the next operation on that file will be read rather than write or unlink. We call this property "entropy" and find it to be very strong in our trace data. Entropy takes effect very quickly; in the DEC trace data for any arbitrary file there is about a 10% chance that the next operation on that file will overwrite it, but once the file has been opened for reading at least once, this drops to less than 4%. After three reads, this becomes just 0.5%, and after 10 reads, less than .001%. In other words, files tend to become read-only once they have survived the first few reads. Surprisingly, this remains true (with slightly shifted curves) even when we restrict ourself to looking at system files (/usr, for example) or user's home directories. File entropy is a very useful property to exploit in a caching strategy, since files that are unlikely to change are the best candidates for replication. This property allows us to identify these files using a small history and without requiring the use or type of a file to be known in advance. It also suggests that the very files that change only infrequently also account for a large proportion of reads; in the trace data, over 66% of reads were of files that had been read at least 6 times in the past.

Figure 3 plots the number of previous opens for reading on a file for each open for reading and each open for writing. Figure 4 is the same plot but restricted to operations on files in users' home directories, which are more volatile. Note that these plots are cumulative; the number on the horizontal axis indicates that there have been that many *or more* reads of the file in the past.

It is worth noting that (over all files) reads outnumber writes by about 5 to 1, and that most reads are of files that have already been read several times before. Most writes, on the other hand, are of files that have been read zero or one times before.

Clearly, locality, entropy, and inertia plus the previously known properties of Unix files suggest that there is much to be gained by caching in distributed file systems. The next section of this paper develops and compares several simple strategies for maintaining client caches.

3. Trace-Driven Simulation of Caching Strategies

The properties described in the previous section suggest that most server traffic for reads can be avoided with a relatively simple client caching algorithm. This section describes a trace driven simulation of several caching schemes to determine whether, indeed, this is true, and to shed some light on the question of a reasonable cache size. We examined and evaluated fairly simple algorithms in order to avoid basing our conclusions on very specific "offline" properties of the trace data. In evaluating the various algorithms, we considered only the number of messages sent between client and server. We did not consider any (possibly non-trivial) local processing required at any CPU.

3.1. Simulation Caching Model

We used the DEC trace data to conduct our simulation. Again, we looked at only opens and unlinks and assumed that any write or unlink renders all cached copies of the file invalid. Each client can keep a cached copy of any file it reads but the file server can assign a (possibly infinite) expiration time to the

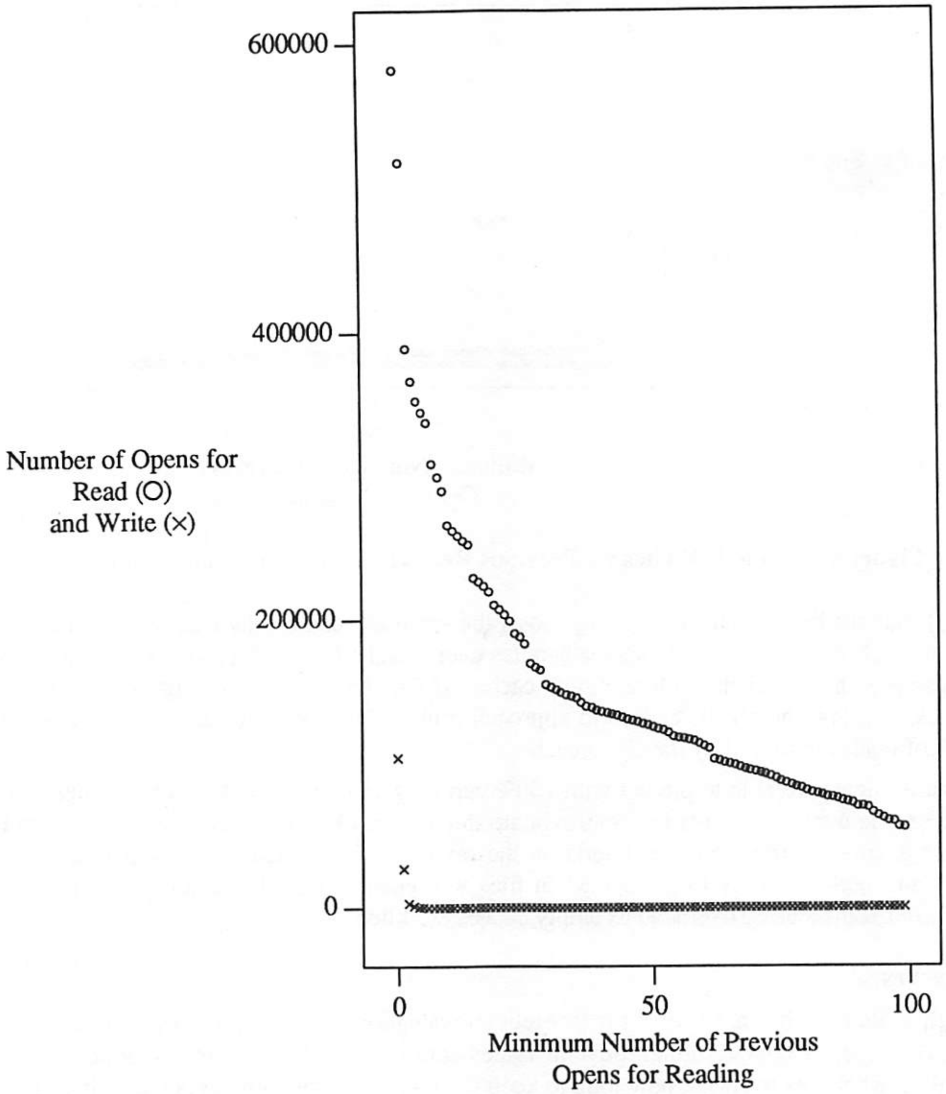


Figure 3 - Reads & Writes vs. Previous Reads (on same file) - All files

client copy at the time it is created. The "owner" of the file (the file server) maintains a list of clients with unexpired copies of each file. If the file is unlinked or written to, each client with an unexpired copy of the file must be notified (it need not be sent the new data, however). If a client reads a file that is in the local cache it need not communicate with the server. Clients may also delete files from their caches at any time without notifying the server. This is an extension of the "stashing" model [10]. Since we are looking only at open operations, we assume that each read is of the entire file; this is probably a reasonable assumption, based on what is known about most Unix file accesses.

3.2. Infinite-Size Caches

This is the simplest case to analyze, and it provides a useful upper bound on the impact of a cache on network traffic. In the infinite cache model, each client keeps a copy of every file it reads until it is told to delete it by the server when the file is written to or unlinked. The results of this trace were very encouraging and suggest that, in fact, caches can have a dramatic impact on network traffic. Of the 686991 operations traced over a five day period, 582302 were read operations, of which 523121 were of valid files in the clients' caches and hence would not require server intervention. So server reads are reduced by 89%. However, 55546 "cancel" messages had to be sent to notify clients of invalid data. If we assume (very

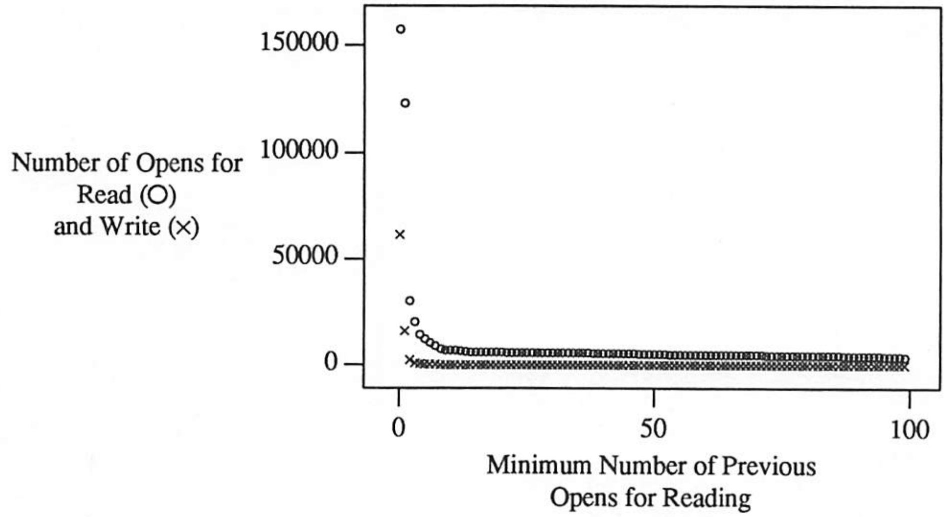


Figure 4 - Reads & Writes vs. Previous Reads (on same file) - /udir Only

conservatively) that sending a "cancel" message costs the same as sending the file, we avoid 80% of the server traffic in a scheme where the file server handles every read. This includes overhead caused by the simulation starting with each client with an empty cache. If the simulation is run for one day before collecting statistics, to allow the clients caches to approach a more "steady" state, the savings is even more dramatic: 93% of reads are served by the client cache.

Note that if clients are able to predict with 100% certainty when a file will be overwritten, the "cancel" messages can be avoided; clients can approximate this if they choose an accurate expire time for the file. This gives us an encouraging upper bound on the savings to be gained by caching in a DFS. With infinite caches and perfect ability to predict when files will change, our simulation yields a 93% client cache hit rate, after some startup overhead as empty caches are filled.

3.3. Finite Caches

Although infinite caches are useful for theoretical evaluation of the impact of caching on network traffic, real caches are, of course, finite. Several issues confront the designer of a practical DFS cache scheme, including what files to cache, how long to keep them in the cache, how to decide what files to discard when the cache is full, and, of course, how big to make the caches.

We assume that each client cache can hold a fixed number of files of arbitrary size each. Although this is not as useful as a simulation of caches of fixed block size, our trace data did not lend itself to this sort of analysis. The average file size of files in the trace was about 11k, so one could approximate the real cache size accordingly.

We considered three finite cache algorithms, and we simulated each using the DEC data. We ran each simulation twice: once gathering statistics for the entire period of the trace, and again where statistics were gathered after it was run for the first simulation day (the "steady state"). For each algorithm, we give the cache sizes simulated, the absolute number of reads not served by client caches and the percentage of total reads this represents, the number of "overhead" messages entailed by the algorithm, the sum of the file traffic and other traffic, and the percentage of total traffic this represents compared with an uncached scheme, assuming that the cost of an "overhead" message and sending a file are equal.

The first algorithm we examined is simple LRU, where, just as in the infinite cache case, the server maintains (along with the usual inode data) a list of all clients that have read the file since it was last written. Clients maintain a simple queue of files, putting the most recently read file at the head and deleting the least recently used files as the cache fills (without notifying the server). When a file is written or unlinked, each client which has made a copy is notified (whether or not the file is still in the client cache). This simple algorithm does fairly well, certainly compared with no caching at all, but still entails the overhead of

sending "cancel" messages to clients who have long flushed the file from their caches. Table 1 is a summary of the LRU algorithm with various cache sizes; 1a is over the whole simulation, and 1b is the "steady state" after one day.

Cache Size	Cache Misses	Miss Rate	Cancel Messages	Total Traffic	Percent of Traffic
0	582302	100.0	55546	637848	109.5
32	208084	35.7	55546	263630	45.3
64	160492	27.6	55546	216038	37.1
128	124841	21.4	55546	180387	31.0
192	109646	18.8	55546	165192	28.4
256	100493	17.3	55546	156039	26.8
320	93006	16.0	55546	148552	25.5
384	87415	15.0	55546	142961	24.6
448	82378	14.1	55546	137924	23.7
512	78605	13.5	55546	134151	23.0
infinite	59181	10.2	55546	114727	19.7

1a - LRU (full trace)

Cache Size	Cache Misses	Miss Rate	Cancel Messages	Total Traffic	Percent of Traffic
0	451029	100.0	40951	491980	109.1
32	155109	34.4	40951	196060	43.5
64	117350	26.0	40951	158301	35.1
128	88349	19.6	40951	129300	28.7
192	74935	16.6	40951	115886	25.7
256	66948	14.8	40951	107899	23.9
320	60064	13.3	40951	101015	22.4
384	54951	12.2	40951	95902	21.3
448	50218	11.1	40951	91169	20.2
512	46757	10.4	40951	87708	19.4
infinite	28227	6.3	40951	69178	15.3

1b - LRU (steady state)

Table 1 - LRU Simulation

The second algorithm we examined attempts to remedy the problems of sending expire messages to clients for files in which they are no longer interested and maintaining the server records for these "dormant" clients. We expire files just before we expect them to be overwritten, freeing the server from the obligation to notify the client of changes once the expire time has occurred. This is somewhat like a generalization of the "Lease" concept [6]. Clients first flush expired files from their caches when selecting a cached file for replacement; if no unexpired files exist, LRU is used.

This, of course, raises the question of how to calculate the expire time. We want to assign long expire times to files that are likely to be used again at the client, and short ones to files that are not likely to be read at the client before being overwritten. Recall that the likelihood of a file being overwritten decreases as the file is read more and more. After considerable experimentation, we found a reasonable expire time to be

$$t + \frac{d}{32} r^2$$

where t is is the current time, d is the length of time since the file was last written, and r is the number of

times the file was read at the server since last written. This reduced the number of cancel messages dramatically (by a factor of almost 40), but also caused a modest decrease in the client cache hit rate (some files expired while still in the client cache and were later read by the client). Table 2 gives the simulation results for EXPIRE.

Cache Size	Cache Misses	Miss Rate	Cancel Messages	Total Traffic	Percent of Traffic
0	582302	100.0	1546	583848	100.3
32	219357	37.7	1546	220903	37.9
64	174292	29.9	1546	175838	30.2
128	140968	24.2	1546	142514	24.5
192	127072	21.8	1546	128618	22.1
256	119098	20.5	1546	120644	20.7
320	112884	19.4	1546	114430	19.7
384	106994	18.4	1546	108540	18.6
448	102360	17.6	1546	103906	17.8
512	99698	17.1	1546	101244	17.4
infinite	88779	15.2	1546	90325	15.5

2a - EXPIRE (full trace)

Cache Size	Cache Misses	Miss Rate	Cancel Messages	Total Traffic	Percent of Traffic
0	451029	100.0	1071	452100	100.2
32	163835	36.3	1071	164906	36.6
64	128119	28.4	1071	129190	28.6
128	100374	22.3	1071	101445	22.5
192	88136	19.5	1071	89207	19.8
256	80943	17.9	1071	82014	18.2
320	75207	16.7	1071	76278	16.9
384	69568	15.4	1071	70639	15.7
448	65328	14.5	1071	66399	14.7
512	62809	13.9	1071	63880	14.2
infinite	52353	11.6	1071	53424	11.8

2b - EXPIRE (steady state)

Table 2 - EXPIRE Simulation

Our final algorithm is a simple adjustment to the EXPIRE algorithm. Clients do not flush expired files from their caches first; all replacement is done by strict LRU. When a cached but expired file is read, the client verifies with the server that the file is unchanged; if so, the rest of the read is processed locally and a new expire time is assigned to the file. This increases the number of locally processed reads to the level of LRU at the expense of these additional messages, while still maintaining the reduced cancel message overhead of EXPIRE. Table 3 gives the simulation results for EXP-LRU.

It is worth noting that whether EXPIRE or EXP-LRU is preferable depends upon the relative cost of sending entire files against the cost of expire and verify messages. If all messages cost about the same (as they would be where files are small and network packets are large), EXPIRE is the clear winner. Figure 5 compares (in the steady state) the total number of messages used by LRU, EXPIRE and EXP-LRU assuming that all messages are of equal cost. If, on the other hand, sending whole files costs considerably more than sending small, constant size, expire and verify messages, EXP-LRU becomes more attractive. Figure 6 compares the three algorithms where sending files is 10 times the cost of other kinds of messages. Of course, a better expire time calculation formula could make EXPIRE the better choice regardless of the

Cache Size	Cache Misses	Miss Rate	Cancel Messages	Verify Messages	Total Messages	Total Traffic	Percent of Traffic
0	582302	100.0	1546	0	1546	583848	100.3
32	208084	35.7	1546	21770	23316	231400	39.7
64	160492	27.6	1546	23779	25325	185817	31.9
128	124841	21.4	1546	25434	26980	151821	26.1
192	109646	18.8	1546	26267	27813	137459	23.6
256	100493	17.3	1546	26774	28320	128813	22.1
320	93006	16.0	1546	27255	28801	121807	20.9
384	87415	15.0	1546	27579	29125	116540	20.0
448	82378	14.1	1546	27907	29453	111831	19.2
512	78605	13.5	1546	28147	29693	108298	18.6
infinite	59181	10.2	1546	29598	31144	90325	15.5

3a - EXP-LRU (full trace)

Cache Size	Cache Misses	Miss Rate	Cancel Messages	Verify Messages	Total Messages	Total Traffic	Percent of Traffic
0	451029	100.0	1071	0	1071	452100	100.2
32	155109	34.4	1071	17313	18384	173493	38.5
64	117350	26.0	1071	18865	19936	137286	30.4
128	88349	19.6	1071	20208	21279	109628	24.3
192	74935	16.6	1071	20956	22027	96962	21.5
256	66948	14.8	1071	21416	22487	89435	19.8
320	60064	13.3	1071	21866	22937	83001	18.4
384	54951	12.2	1071	22177	23248	78199	17.3
448	50218	11.1	1071	22485	23556	73774	16.4
512	46757	10.4	1071	22723	23794	70551	15.6
infinite	28227	6.3	1071	24126	25197	53424	11.8

3b - EXP-LRU (steady state)

Table 3 - EXP-LRU Simulation

relative cost of messages.

4. Consistency in Unreliable Networks

The above algorithms work well in a perfect network that remains fully connected and in which no hosts ever fail. Of course, this is even less true of the large networks for which such strategies are designed than it is of small, local networks. Clearly, consistency is an issue in such systems. Although it is well beyond the scope of this paper to conduct an in-depth analysis of consistency in distributed systems, it is worth discussing a few issues that naturally arise out of file replication schemes such as those we have described.

First, we must define what we mean by consistency. At one end of the spectrum, we have what is often called "Unix semantics": each update is guaranteed to be reflected by all other processes as soon as it completes. Clearly, strict Unix semantics are expensive to maintain in a DFS, since it requires clients to verify that their caches are up to date with the server for every read. This would all but eliminate the benefits of caching. Furthermore, if the connection to the server is down, the client read always fails, even when file is in the local cache. Few DFSs attempt to guarantee complete Unix semantics, however, and even some uniprocessor systems do not provide this degree of consistency. At the other end of the spectrum, we could guarantee nothing, and if a client looses its connection to the server, allow it to process

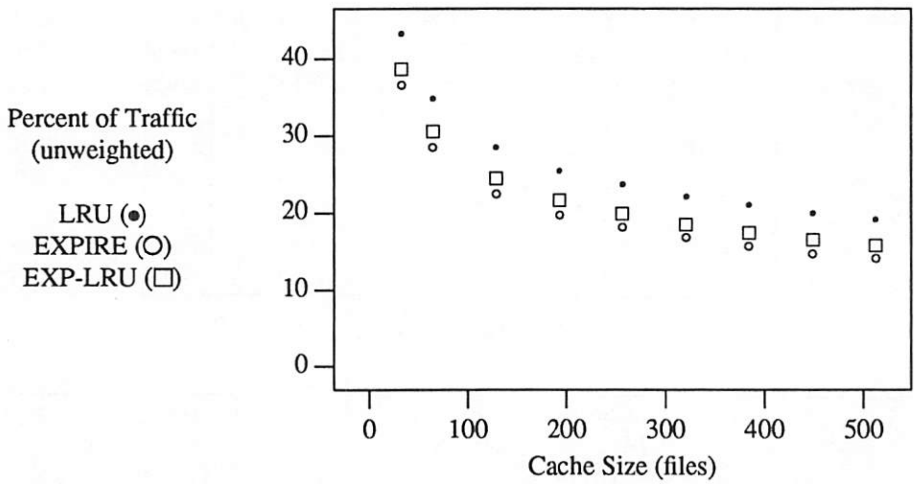


Figure 5 - Cache Algorithms Compared: File Cost == Message Cost

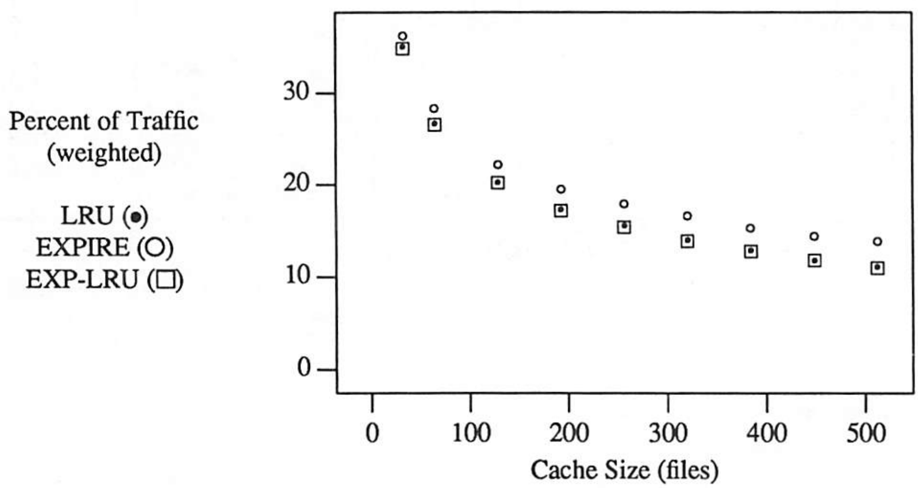


Figure 6 - Cache Algorithms Compared: File Cost == 10x Message Cost

requests with its cache for any files it chooses.

Again, we seek a middle ground between these two extremes in which clients can have some guarantees that the files they read are up to date, but do not have to verify each read with the server. The problem is for clients to determine that they have not lost contact with the file server (and missed invalidation messages) without having to contact the server for each read. We identify two consistency constraints that can be maintained without excessive server interaction. The first, which we call "bounded currency", guarantees that all reads from the cache reflect data out of data by no more than some real time bound. This is simple to implement with "keepalive" messages. Nothing is guaranteed about the self-consistency of the reads and writes, however. The second, "one-copy serializability", states that the global sequence of reads and writes must correspond to some ordering of events if all machines were connected to a single file system. Note that this says nothing about whether a read reflects a recent copy, but just that the set of reads and writes must be self-consistent.

4.1. Bounded Currency

Bounded currency guarantees that cached copies were up to date as of some known time in the past. This can be implemented by flushing the cache after some period of time (and in fact, this is how NFS provides the degree of consistency that it does). Less drastically, each client can keep track of the last time it communicated with each server from which it caches files. When a file in the local cache is read, the client first verifies that it has communicated with the server within the specified time bound. If it has not, it initiates a "keepalive" exchange with that server before completing the read. If the exchange fails, the read cannot be completed.

Clearly, this entails overhead of at most one exchange per client/server pair every time period, regardless of the the number of files each client has from each server. If the clients communicate with the server for other reasons, the number of overhead messages is lowered, of course.

4.2. One-Copy Serializability

This is a generalization of the well-known consistency metric from the field of distributed databases. Simply put, it requires that the global sequence of reads and writes must be self-consistent, in that they must correspond to some sequence of reads and writes on a single file system. This, by itself, does not require that a read actually return data that is in any sense recent. For example, if a client reads a file from a server, makes a cached copy, and becomes partitioned from the server, it might continue to use the old cached copy long after the file has been overwritten. However, we are guaranteed that any other files we read were not created based on data that is unavailable to us.

It is actually fairly easy to maintain one-copy serializability without excessive network traffic. It is sufficient to guarantee that each read from a client's cache reflects data that is current as of the last time that client wrote data read by remote machines. A simple (and fairly inexpensive) way to maintain this constraint is for a clients to verify that it can communicate with the file server any time it reads a file that was cached prior to the last time it wrote to a remotely readable file. If the client has already communicated with the server since it last wrote to a remote file, it need not initiate the keepalive exchange; it need only do so when it has not heard from the server since the last such write.

Observe that the two constraints are complementary; one provides temporal guarantees while the other provides logical guarantees. In practice, it is probably reasonable to provide a combination of bounded currency and one-copy serializability; note that the overhead keepalive messages required to maintain one constraint can reduce the number required by the other. It is also worth noting that not all clients actually require a high degree of consistency, and such parameters as the frequency of required keepalive messages could be a tunable option specified at either open or mount time.

5. Future Work - Cache Hierarchies

While the simulations described above suggest that small client caches can reduce server traffic by a factor of about five, this is still not enough to allow file systems on the massive scale we seek. As a distributed file system grows beyond a small set of machines on a single network, it is advantageous to organize clients into hierarchies so that they do not all contact the file server directly. In such a structure cached (or replicated) copies are used not only by the machine with the cache, but by "child" machines as well. A hierarchical structure offers potential performance benefits to both client and server. From the server's perspective, it can serve orders of magnitude more clients while communicating with only a few top-level nodes. At the same time, clients have access to potentially large caches that may be fewer network hops away.

Some experimental systems [2] permit such a organization to be specified statically as the system is set up. While such schemes are potentially very useful for large homogeneous systems belonging to a single organizational entity, we believe that a more dynamic scheme is required for very large scale file systems. In particular, static hierarchies are inefficient for files that are read infrequently (since they introduce extra layers between client and server) and may cause bottlenecks when the exact hierarchy in place does not reflect the actual traffic load (as when clients in different "leaves" share files). It is difficult to predict the access patterns of a network in advance, and it only becomes more difficult as the network grows and

includes machines about which less is known. Our current work focuses on designing a file system that constructs an adaptive hierarchy for each file, so that heavily used files have several layers of caches between clients and servers, while all clients can go to the server for lightly used files. Clients should be able "share" caches on neighboring machines (especially when the server is very distant) and servers should be able off-load frequent file accesses to machines with surplus cache space. Such a scheme could, for example, be appropriate for very large networks such as the Internet, where no central administrative control exists and machines come and go at fairly random intervals.

We are presently addressing the question of how to best construct such a dynamic hierarchy. For example, each server could put a bound on the number of clients to which it is willing to provide cached copies. When the maximum number of clients is reached for a particular file, it could send the list of machines with copies to new clients requesting the file. The new client could select the "best" machine on the list (according to proximity or some other criteria) and route all future requests through that machine. This could be repeated recursively when the client machines themselves become overloaded, and there could be mechanisms for allowing machines to shift the caching load to one another.

6. Summary and Conclusions

A key requirement for successful large scale DFSs is the ability of clients to hide their file system activity from their servers. Caches, in the form of a modest amount of local persistent storage, provide a convenient and well-understood mechanism to do this. We have identified several properties (inertia, entropy and locality) present in a trace of workstation file accesses that suggest that caching can be used to considerable advantage in a DFS.

A trace driven simulation of simple caching schemes further supports the notion that server traffic can be reduced dramatically with fairly small caches. A trace of one such scheme showed that if clients can cache 200 files, 80% of network traffic is eliminated. As memory becomes less expensive, it will become practical to provide very fast file system caches of this size in memory.

Although the problem of consistency in replicated distributed systems is a complex one, two simple constraints (bounded currency and one-copy serializability) provide a fairly high degree of consistency at a reasonable cost in overhead. These can be combined and tuned to provide the degree of consistency required at any particular client.

In summary, we believe that, given small client caches and fairly simple caching algorithms such as those we have described, file servers (and the networks to which they are attached) can serve many more clients than possible with conventional methods.

7. Acknowledgments

We are extremely grateful to Andy Hisgen at DEC SRC for making the trace data available to us. The following people at DEC SRC contributed to the file system tracing facility: Susan Owicki, B. Kumar, Jim Gettys, Deborah Hwang, and Andy Hisgen. The actual trace data was gathered by Andy Hisgen. Brad Barber made helpful comments on drafts of this paper.

8. References

[1] Blaze, M. "Issues in Massively Distributed File Systems." *Proc. 2nd Princeton University SystemsFest,* April, 1990.

[2] Siegel, A., Birman, K., & Marzullo, K. "Deceit: A Flexible Distributed File System." *TR 89-1042,* Dept. Comp. Sci., Cornell University, Nov. 1989.

[3] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., & Lyon, B. "Design and Implementation of the Sun Network File System." *Proc. USENIX,* Summer, 1985.

[4] Ousterhout J., et al. "A Trace-Driven Analysis of the Unix 4.2 BSD File System." *Proc. 10th ACM Symp. Op. Sys. Principles,* 1985.

[5] Staelin, C. "File Access Patterns" *CS-TR-179-88* Dept. Comp. Sci, Princeton U, 1988.

[6] Gray, C. & Cheriton, D. "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency." *Proc, 12th ACM Symp. Op. Sys. Principles,* December, 1989.

[7] Floyd, R. "Short-Term File Reference Patterns in a UNIX Environment," *TR-177* Dept. Comp. Sci, U. of Rochester, 1986.

[8] Guy, R., et al. "Implementation of the Ficus Replicated File System," *Proc. Summer 1990 USENIX,* 1990.

[9] Kazar, M., et al. "DEcorum File System Architectural Overview," *Proc. Summer 1990 USENIX,* 1990.

[10] Alonso, R., Barbara, D., and Cova, L. "Using Stashing to Increase Node Autonomy in Distributed File Systems," *Proceedings of the Ninth Symposium on Reliable Distributed Systems,* October 9-11, 1990, Huntsville, Alabama.

Matt Blaze

Matt Blaze is a Ph.D. candidate in Computer Science at Princeton University, where he expects to receive his degree in the Spring of 1992. His research interests include distributed systems, operating systems, databases, and programming environments. His current work focuses on developing a highly scalable distributed file system. In 1990 he was awarded a USENIX Graduate Scholarship. In 1988 he received an M.S. in Computer Science from Columbia University and in 1986 a B.S. from Hunter College. He can be reached via email at mab@princeton.edu or via US mail at Dept. of Computer Science, Princeton University, 35 Olden Street, Princeton NJ 08544.

Rafael Alonso

Rafael Alonso received a B.A. in Mathematics and Computer Science from New York University, an M.S. in Electrical Engineering from Columbia University, and a Ph.D. in Computer Science from U.C. Berkeley. He is currently an Assistant Professor in the Department of Computer Science at Princeton University. His areas of interest are distributed computing and databases. His most recent topics of research have been distributed file systems and heterogeneous databases.

Management of Replicated Volume Location Data in the Ficus Replicated File System*

Thomas W. Page Jr., Richard G. Guy, John S. Heidemann, Gerald J. Popek[†]
Wai Mak, and Dieter Rothmeier

Department of Computer Science
University of California Los Angeles
{page,guy,popek,johnh,waimak,dieter}@cs.ucla.edu

Abstract

Existing techniques to provide name transparency in distributed file systems have been designed for modest scale systems, and do not readily extend to very large configurations. This paper details the approach which is now operational in the Ficus replicated UNIX filing environment, and shows how it differs from other methods currently in use. The Ficus mechanism permits optimistic management of the volume location data by exploiting the existing directory reconciliation algorithms which merge directory updates made during network partition.

1 Introduction

Most UNIX file system implementations separate the maintenance of name binding data into two levels: within a volume (or filesystem), directory entries bind names to low-level identifiers (such as inodes); a second mechanism is used to form a super-tree by "gluing" the set of volumes to each other. The traditional UNIX volume super-tree connection mechanism has been widely altered or replaced to support both small and large scale distributed file systems. Examples of the former are Sun's Network File System (NFS) [13] and IBM's TCF [12]; larger scale file systems are exemplified by AFS [7], Decorum [6], Coda [14], Sprite [9], and Ficus [2, 10]).

The problem addressed by this paper is simply stated as follows: in the course of expanding a path name in a distributed file system, the system encounters a *graft point.* That is, it reaches a leaf-node in the current volume which indicates that path name expansion should continue in the root directory of another volume which is (to be) grafted on at that point. How does the system identify and locate (a replica of) that next volume? Solutions to this problem are very much

*This work was sponsored by DARPA contract number F29601-87-C-0072.
[†]This author is also associated with Locus Computing Corporation.

constrained by the number of volumes in the name hierarchy, the number of replicas of volumes, the topology and failure characteristics of the communications network, the frequency or ease with which replica storage sites or graft points change, and the degree to which the hierarchy of volumes spans multiple administrative domains.

1.1 Related Solutions

The act of gluing sub-hierarchies of the name space together is commonly known as *mounting*.[1] In a conventional single-host UNIX system, a single mount table exists which contains the mappings between the mounted-on leaf nodes and the roots of mounted volumes. However, in a distributed file system, the equivalent of the mount table must be a distributed data structure. The distributed mount table information must be replicated for reliability, and the replicas kept consistent in the face of update.

Most distributed UNIX file systems to some degree attempt to provide the same view of the name space from any site. Such *name transparency* requires mechanisms to ensure the coherence of the distributed and replicated name translation database. NFS, TCF, and AFS each employ quite different approaches to this problem.

To the degree that NFS achieves name transparency, it does so through convention and the out-of-band coordination by system administrators. Each site must explicitly mount every volume which is to be accessible from that site; NFS does not traverse mount points in remotely mounted volumes. If one administrator decides to mount a volume at a different place in the name tree, this information is not automatically propagated to other sites which also mount the volume. While allowing sites some autonomy in how they configure their name tree is viewed as a feature by some, it leads to frequent violations of name transparency which in turn significantly complicates the users' view of the distributed file system and limits the ability of users and programs to move between sites. Further, as a distributed file system scales across distinct administrative domains, the prospect of maintaining global agreement by convention becomes exceedingly difficult.

IBM's TCF, like its predecessor Locus [12], achieves transparency by renegotiating a common view of the mount table among all sites in a partition every time the communications or node topology (partition membership) changes. This design achieves a very high degree of network transparency in limited scale local area networks where topology change is relatively rare. However, for a network the size of the Internet, a mount table containing several volumes for each site in the network results in an unmanageably large data structure on each site. Further, in a nationwide environment, the topology is constantly in a state of flux; no algorithm which must renegotiate global agreements upon each partition membership change may be considered. Clearly neither of the above approaches scales beyond a few tens of sites.

Cellular AFS [15] (like Ficus) is designed for larger scale application. AFS employs a *Volume Location Data Base* (VLDB) for each cell (local cluster) which is replicated on the cell's backbone servers. The mount point itself contains the cell and volume identifiers. The volume identifier is used as a key to locate the volume in a copy of the VLDB within the indicated cell. Volume location information, once obtained, is cached by each site. The VLDB is managed separately

[1] We will henceforth use the term "graft" and "graft point" for the Ficus notion of grafting volumes while retaining the mount terminology for the UNIX notion of mounting filesystems.

from the file system using its own replication and consistency mechanism. A primary copy of the VLDB on the *system control machine* periodically polls the other replicas to pull over any updates, compute a new VLDB for the cell, and redistribute it to the replicas. The design does not permit volumes to move across cell boundaries, and does not provide location transparency across cells, as each cell's management may mount remote cell volumes anywhere in the namespace. Again, this may be billed as a feature or a limitation depending on where one stands on the tradeoff between cell autonomy and global transparency.

1.2 The Ficus Solution

Ficus uses AFS-style *on disk mounts*, and (unlike NFS) readily traverses remote mount points. The difference between the Ficus and AFS methods lies in the nature of Ficus volumes (which are replicated) and the relationship of graft points and volume location databases.

In Ficus, like AFS [5], a volume is a collection of files which are managed together and which form a subtree of the name space[2]. Each logical volume in Ficus is represented by a set of volume replicas which form a maximal, but extensible, collection of containers for file replicas. Files (and directories) within a logical volume are replicated in one or more of the volume replicas. Each individual volume replica is normally stored entirely within one UNIX disk partition.

Ficus and AFS differ in how volume location information is made highly available. Instead of employing large, monolithic mount tables on each site, Ficus fragments the information needed to locate a volume and places the data in the mounted-on leaf (a graft point). A graft point maps a set of volume replicas to hosts, which in turn each maintain a private table mapping volume replicas to specific storage devices. Thus the various pieces of information required to locate and access a volume replica are stored where they will be accessible exactly where and when they will be needed.

A graft point may be replicated and manipulated just like any other object (file or directory) in a volume. In Ficus the format of a graft point is compatible with that of a directory: a single bit indicates that it contains grafting information and not file name bindings. The extreme similarity between graft points and normal file directories allows the use of the same optimistic replication and reconciliation mechanism that manages directory updates. Without building any additional mechanism, graft point updates are propagated to accessible replicas, uncoordinated updates are detected and automatically repaired where possible, and reported to the system administrators otherwise.

Volume replicas may be moved, created, or deleted, so long as the target volume replica and any replica of the graft point are accessible in the partition (one copy availability). This optimistic approach to replica management is critical, as one of the primary motivations for adding a new volume replica may be that network partition has left only one replica still accessible, and greater reliability is desired.

This approach to managing volume location information scales to arbitrarily large networks,

[2] Whereas a filesystem in UNIX is traditionally one-to-one with a disk partition, a volume is a logical grouping of files which says nothing about how they are mapped to disk partitions. Volumes are generally finer granularity than filesystems; it may be convenient to think of several volumes within one filesystem (say one volume for each user's home directory and sub-tree) though the actual mapping of volumes to disk partitions is a lower level issue.

with no constraints on the number of volumes, volume replicas, changes in volume replication factors, or network topology and connectivity considerations.

1.3 Organization of the Paper

This introduction has presented the problem of locating file replicas in a very large scale, geographically distributed replicated file system, and briefly stated our solution. Section 2 gives an overview of Ficus to put our solutions in context. Section 3 details our volume location and autografting strategy. Then, Section 4 examines the consequences of updating volume location information using the optimistic concurrency and lazy propagation policy. Section 5 presents our conclusions.

2 Overview of Ficus

In order to convey the context in which our solutions to replicated volume location information management are employed, this section presents an overview of the Ficus replicated file system. Both its optimistic concurrency control and stackable layered structuring significantly influence the character of our solutions to configuring the name space.

Ficus is a replicated distributed file system which can be added easily to any versions of the UNIX operating system supporting a VFS interface. It is intended to provide highly reliable, available, and scalable filing service, both to local clusters of machines and to geographically distributed work groups. It assumes no limits on the number of sites in the network, the number of Ficus volumes, or on the number of volume replicas. However, while placing no hard limit on the number of replicas, Ficus does assume that there is seldom call for more than a small number of fully updatable replicas of any one volume and hence optimizes for the limited case.

Ficus supports very high availability for write, allowing uncoordinated updates when at least one replica is accessible. *No lost updates* semantics are guaranteed; conflicts are reliably detected and directory updates are automatically reconciled. Asynchronous update propagation is provided to accessible copies on a "best effort" basis, but is not relied upon for correct operation. Rather, periodic *reconciliation* ensures that, over time, all replicas converge to a common state. We argue that serializability is not provided in single machine UNIX file systems, and is not required in distributed file systems. This kind of policy seems necessary and appropriate for the scale and failure modes in a nation-wide file system. Details of the reconciliation algorithms may be found in [3, 11], while for more information about the architecture of Ficus see [10, 4].

2.1 Reconciliation

The essential elements of the optimistic replica consistency strategy are the reconciliation algorithms which ensure eventual mutual consistency of replicas. They are necessary since update is permitted during network partition, and since update propagation to accessible replicas is not

coordinated via a two-phase commit protocol. The algorithms guarantee that updates will eventually propagate and that conflicting updates will be reliably detected, even given the complex topology and failure modes of the internet communications environment. The reconciliation algorithms propagate information through intermediate sites in a "gossip" fashion to ensure that all replica sites learn of updates, even though any two replica hosts may rarely or never directly communicate with each other.

The optimistic consistency philosophy applies not only to updates to existing files, but also to the name space (creation and deletion of names and the side-effects of deleting the last name for a file). The Ficus reconciliation system is able to reconcile most "conflicting" directory updates since the semantics of operations on directories are well-known. The exception is the independent creation of two files with the same name (a name conflict) and the update of a file which is removed in another replica (a remove-update conflict) which the system can only detect and report. In the case of operations on arbitrary files where the system cannot know the semantics of the updates, Ficus guarantees only to detect and report all conflicts.

A reconciliation daemon running the algorithms periodically walks the tree of a volume replica, comparing the version vector of each file or directory with its counterpart in a remote replica of the volume. Directories are reconciled by taking the union of the entries in the two replicas less those that have been deleted. Consider for example, the case where two replicas, `A` and `B`, of a directory have been updated independently during a network partition. A new file name, `Brahms`, has been added to `A`, while the name, `Bach`, has been deleted from `B`. When these two replicas reconcile, it is clear that neither can be directly propagated over the other because an update-update conflict exists (both directory replicas have changed). However, given the simple semantics of directories, it is clear that the correct merging of the two replicas should contain the new name `Brahms`, and not the name `Bach`. This is what results in Ficus. The case where one replica knows about a file name create and the other does not is disambiguated from the case where one has the file's entry but the other has deleted it by marking deleted entries as "logically deleted". File names marked logically deleted may be forgotten about when it is known that all replicas know that all replicas have the name marked logically deleted (hence the two-phased nature of the algorithms). When the link count for a replica goes to zero, garbage collection may be initiated on the pages of the file; however, the file data cannot actually be freed until reconciliation determines that all replicas have a zero link count (no new names have been created), a dominant version exists, and it is the dominant replica being removed (there is no remove-update conflict).[3]

Each file replica has a version vector attached to it with a vector element for each replica of the file. Each time a file is updated, the version vector on the copy receiving the update is incremented. File replicas are reconciled by comparing their version vectors as described in [11]. As a part of reconciliation, a replica with a dominant version vector is propagated over the out-of-date replica which also receives the new version vector. If neither replica's vector is dominant, a write-write conflict is indicated on the file and a conflict mark is placed on the local replica, blocking normal access. Access to marked files is permitted via a special syntax in order to resolve conflicts. We will see how these same algorithms may be leveraged to manage the volume location data.

[3] The details of the two-phase algorithms for scavenging logically deleted directory entries and garbage collecting unreferenced files can be found in [1, 3]. The issues are somewhat more subtle than a first glance would suggest.

2.2 Stackable Layers

Ficus is also unique in its support for extensible filing environment features via stackable layered structuring. This architecture permits replication to co-exist with other independently implemented filing features and to remain largely independent of the underlying file system implementation. We briefly introduce the layered architecture here; for more details, see [4, 10, 2]

The stackable layers architecture provides a mechanism whereby new functionality can be added to a file system transparently to all other modules. This is in contrast to today's UNIX file systems in which substantial portions must be rebuilt in order to add a new feature. Each layer supports a symmetrical interface for both calls to it from above and with which it performs operations on the layers below. Consequently, a new layer can be inserted anywhere in the stack without disturbing (or even having source code to) the adjacent layers. For example, a caching layer might look for data in a cached local file, forwarding reads to the underlying layers representing a remote file in the case of a cache miss. Thus, stackable layers is an architecture for extensible file systems.

Each layer supports its own vnode type, and knows nothing about the type of the vnode stacked above or below[4]. The vnodes, which represent the abstract view of a given file at each layer, are organized in a singly linked list. The kernel holds a pointer to the head of the list only. Operations on the head of the list may be performed by that vnode, or forwarded down the stack until they reach a layer which implements them. At the base of the stack is a layer with no further descendants, typically a standard UNIX file system.

The term "stack" is somewhat of a misnomer as there may be both fan-out and fan-in in a stack. Fan-in occurs when a vnode is part of more than one stack (for example when a file is open concurrently on more than one site). Fan-out occurs when a single file at a higher layer is supported by several files at the next layer down. For example, in Ficus replication, a single file from the user's point of view is actually supported by several file replicas, each with its own underlying stack.

Replication in Ficus is implemented using two cooperating layers, a "logical" and a "physical" layer. The logical layer provides layers above with the abstraction of a single copy, highly available file; that is, the existence of multiple replicas is made transparent by the logical layer. The physical layer implements the abstraction of an individual replica of a replicated file. It uses whatever underlying storage service it is stacked upon (such as a UNIX file system or NFS) to store persistent copies of files, and manages the extended attributes about each file and directory entry. When the logical and physical layers execute on different machines, they may be separated by a "transport layer" which maps vnode operations across an RPC channel in a manner similar to NFS. Figure 1 illustrates a distributed file system configured with three replicas: one on a local UNIX box, one connected remotely via a transport layer, and a third stored on an IBM disk farm running MVS connected by NFS (with a UNIX box running the Ficus physical layer acting as a front-end). The replicated file system is shown mounted both on a UNIX workstation and a PC (via PC-NFS).

[4]The exception is when a service such as Ficus replication is implemented as a pair of cooperating layers, a layer may assume that somewhere beneath it in the stack is its partner.

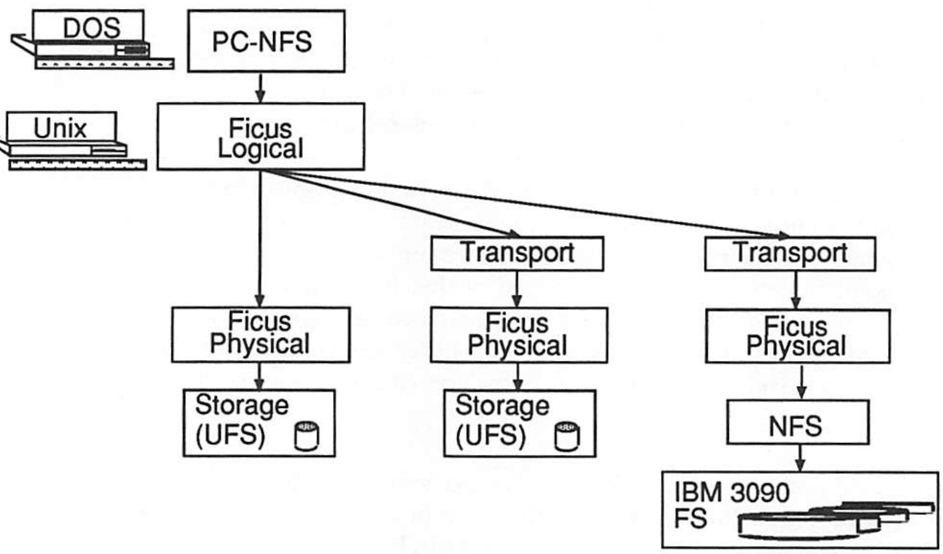


Figure 1: A Ficus layer configuration with three replicas.

2.3 Ficus Layers and Volumes

It is the Ficus logical layer which implements the concept of a volume. Each mounted logical layer supports exactly one logical volume. The Ficus physical layer supports the concept of a volume replica. Mounted under each logical layer is one physical layer per volume replica. The volume location problem concerns how an instance of a logical layer finds the instances of physical layers to attach, and how to move, add, or delete such physical layer instances. Consequently, it is the Ficus logical level which interprets the volume location information contained in graft points; as far as the physical layer is concerned, a graft point is a normal directory.

3 Volume Locating and Autografting

A volume is a self-contained rooted directed acyclic graph of files and directories. A volume replica is represented by a vfs structure (see [8]). Each volume replica must store a replica of the root directory, though storage of any other file or directory replica within the volume is optional.

3.1 Autografting

In a distributed file system which spans the Internet, there may be hundreds of thousands of volumes to which we might desire transparent access. However, any one machine will only ever access a very small percentage of the available volumes. It is therefore prudent to locate and

graft volumes on demand, rather than *a priori*. The main memory data structures associated with grafted volumes which are not accessed for some time may be reclaimed. Hence only those volumes which are actually in use on a given site take up resources on that machine. We refer to this on demand connection of the super-tree as *autografting*.

Since a graft point resides in a "parent" volume, although referring to another volume, the graft point is subject to the replication constraints of the parent volume. There is no a priori requirement that the replication factor (how many replicas and their location in the network) of a graft point match, or even overlap, that of the child volume. If each site which stores a replica of the parent directory in which the graft point occurs also stores a copy of the graft point, the location information is always available whenever the volume is nameable. There is very little benefit to replicating the graft point anywhere else and considerable loss if it is replicated any less.

The graft point object is a table which maps volume replicas to their storage site. The sequence of records in a graft point table is in the same format as a standard directory and hence may be read with the same operations used to access directories. Each entry in a graft point is a triple of the form $\langle volid, replid, hostname \rangle$ identifying one replica of the volume to be grafted. The *volid* is a globally unique identifier for the volume. The *replid* identifies the specific volume replica to which the entry refers. The *hostname* is the internet address of the host which is believed to house the volume replica. The system then uses this information to select one or more of these replicas to graft. If later, the grafted volume replica is found not to store a replica of a particular file, the system can return to this point and graft additional replicas as needed.

In the course of expanding a path name (performing a `vop_lookup` operation on a vnode at the Ficus logical level), the node is first checked to see if it is a graft-point. If it is, and a volume is already grafted as indicated by a pointer to the root vnode of that volume, pathname expansion simply continues in that root directory. The root vnodes may be chained allowing more than one replica of the grafted volume to be grafted simultaneously. If no grafted root vnode is found, the system will select and autograft one or more volume replicas onto the graft point.

In order to autograft a volume replica, the system calls an application-level graft daemon on its site. Each site is responsible for mapping from volume and replica identifiers to the underlying storage device providing storage for that volume. If the *hostname* is local, the graft daemon looks in the file `/etc/voltab` for the path name of the underlying volume to graft. If the *hostname* is remote, the graft daemon obtains a file handle for the remote volume by contacting the remote graft daemon (similar to an NFS mount; see [13]) and completes the graft.

A graft point caches the pointer to the root node of a grafted volume so that it does not have to be reinterpreted each time it is traversed. The graft of a volume replica which is not accessed for some time is automatically pruned so it does not continue to consume resources.

3.2 Creating, Deleting and Modifying Graft Points

Creating and deleting graft point objects require operations on the directory containing the graft point to add or remove a name for the object. Updates to the containing directory which create and delete names for graft points are handled identically to updates creating or deleting any entry

in a directory. Note that moving a graft point is equivalent to creating a copy of it in a different place in the name hierarchy, and deleting the original name. These updates change the way the super-tree of names is configured and hence are very rare.

The graft points themselves (as opposed to the directory containing them) are modified whenever a volume replica is created, dropped, or moved from one host to another. This type of operation is transparent to the user in that it does not affect the name space.

While updating a graft point is also a relatively rare event, when it does occur, it is generally important. Hence it is not reasonable to require that all, or even a majority of the replicas of the graft point be accessible. Further, the motivation for updating a graft point may be at its greatest precisely when the system is unstable or partitioned. Perhaps the whole reason for updating the graft point is to add an additional replica of a volume for which, due to partitions or node failures, only a single replica remains accessible; this update must be permitted, even though it cannot immediately propagate to all replicas of the graft point.

Hence, for exactly the same reason that Ficus utilizes an optimistic philosophy for maintaining the consistency of files and directories, the same philosophy must be applied to graft points. Rather than adopting a separate mechanism for maintenance of super-tree information, Ficus makes double use of its directory management algorithms. This is achieved by structuring graft points with exactly the same format as directories and, as a sequence of records, very similar semantics.

3.3 Update Propagation

All updates in Ficus, whether to files, directories, or graft points, are performed first on a single replica of the object. An update notification is then placed on a queue serviced by a daemon which attempts to send out update notifications to other accessible replicas. The notification message is placed on another queue at the receiving sites. Graft point and directory update notifications contain enough information to apply the update directly to the replica if the version vector attached to the message dominates the local replica. Normal file updates, on the other hand, are not piggybacked with the notification message and must instead be pulled over from an up-to-date replica. Update notification is on a one shot, best effort basis. Any replica which is inaccessible or otherwise fails to receive the notification will be brought up to date later by *reconciling* with another replica.

The reconciliation daemon running on behalf of each volume replica ensures eventual mutual consistency of both the replicated directories containing graft points, and the entries in the replicated tables. It periodically checks (directly or indirectly) every other replica to see if a newer version exists. If a newer version is found, it initiates update propagation; if an update conflict is found, a conflict mark is placed on the object which blocks normal access until the conflict is resolved. Access to marked objects is permitted via a special syntax for use in resolving conflicts.

4 Concurrent Updates to Graft Points

As with directories, the semantics of graft point updates are quite simple, and hence updates which would have to be considered conflicting if viewed from a purely syntactic point of view may be automatically merged to form a consistent result. For example, if in non-communicating partitions, two new replicas of the same volume are created, the two resulting graft point replicas will each have an entry that the other does not have. However, it is clear that the correct merged graft point should contain both entries, and this is what will occur.

The somewhat surprising result is that, unlike directories, there are no conflicts resulting from the optimistic management of the replicated graft point tables that cannot be automatically merged by the existing directory reconciliation algorithms. That is, entries in a graft point experience neither name conflicts or remove-update conflicts. However, as an entry in a directory, the graft point itself (as opposed to the data in the graft point), may be manipulated so as to generate a conflict. The reconciliation algorithms will automatically merge all updates to graft points, and reliable detect any name conflicts or remove-update conflicts on the directory in which the graft point resides.

To understand how the directory reconciliation algorithms double as graft point reconciliators, it is important to know how the graft point entries are mapped into directory entry format. The triple $\langle volid, replid, hostname \rangle$ is encoded in the file name field of the directory entry (the inode pointer field is not used). As a result, graft point semantics are even simpler than directories as name conflicts do not occur. Recall that if two different files are created in different copies of the same directory, a name conflict results, since, in UNIX semantics, names are unique within directories. However, in the case of graft points two independent entries in the table with the same $\langle$ *volid*, *replid*, *hostname* $\rangle$ may be considered equivalent and cause no conflict[5].

A deleted entry in the graft point table (dropped replica) is indicated with the "logically deleted" bit turned on. This is used to distinguish between entries which have been deleted, and entries which never existed. Like directory entries, deleted graft point entries may be expunged when all replicas have the entry logically deleted, and all replicas know that all replicas know the entry is logically deleted (hence the two-phase nature of the reconciliation algorithms; see [3]).

Now consider moving a volume replica. When a replica storage site is changed, the $\langle$ *hostname* $\rangle$ field in the graft point entry is updated. Since this corresponds to a rename (the field changing is part of the name in the directory entry view), it is carried out by creating a new entry and marking the old one as logically deleted. When the two replicas of the graft point table are reconciled, it will be found that a new entry exists and an old entry has been marked deleted; the out-of-date replica will update its graft point accordingly. It cannot occur that the same entry is moved independently in two copies of the graft point, since one needs to be able to access a replica in order to move it[6]. Hence if replica A of the graft point is updated to reflect a change of host for a volume replica, then no other replica of the graft point can update the host field until it has received the update propagation or reconciled with a copy which has, because until then, it will not be able to find the replica.

[5] Due to the way volume ids and replica ids are assigned, it is not possible to create different volumes or replicas independently and have them assigned the same id.

[6] This requires that moving a replica and updating the graft point must be done atomically so that replicas are not lost in the event of a crash.

4.1 Reconciliation of Directories Containing Graft Points

While no conflicts are possible within the replicated volume location tables, the graft points themselves are named objects in a replicated directory which may be subject to conflicting updates. As with uncoordinated updates to any replicated directory in Ficus, the reconciliation algorithms guarantee to merge the inconsistent replicas to create a coherent and mutually consistent version of the directory. The same exceptions also apply; reconciliation will reliably detect name conflicts or remove update conflicts, even if they involve graft points.

A name conflict occurs when in one copy of a directory a new graft point is created, while in another replica, another object (be it a graft point, ordinary file, etc.) is created with the same name. The system saves both entries in the reconciled directory, marking them in conflict, and allowing access by an extended disambiguating name for purposes of resolving the conflict.

A remove-update conflict is created when in one replica the last name for a graft point is deleted (link count becomes zero), while in another the graft point is modified internally (by adding or deleting replica entries). In this case, the graft point disappears from the directory in which it occurs, but is saved in an orphanage from which it may be later retrieved or deleted[7].

5 Conclusions

In order to provide a network transparent view of the file name space, all sites in the system must agree on what volumes are grafted where, and be able to locate a replica. Previous methods which rely on either a fully replicated mount table, informal coordination among system administrators, or replication only within a cell, fail when the system scales towards millions of sites. The solution to this problem in Ficus is predicated on the beliefs that the volume location data must be selectively replicated for performance and availability, and the replicas must be optimistically managed since a conservative approach restricts updates unacceptably.

This paper presents a solution to volume location data management in which the super-tree of volumes is glued together by graft points which are stored in the file hierarchy itself. Graft points may be replicated at any site at which the volume in which they occur is replicated. The same reconciliation daemons which recover from uncoordinated update to files and directories in Ficus also manage the graft point objects, detecting and propagating updates. By structuring graft points internally just like ordinary directories, no modifications are needed to the reconciliation implementation to support graft points. While it has always been our contention that the reconciliation algorithms which were originally developed for the express purpose of maintaining a hierarchical name space were applicable in a wider context, this re-use of the reconciliation daemons unmodified provides the first such evidence.

The graft point solution presented in this paper has been implemented and is operational in Ficus. Volumes are autografted on demand, and ungrafted when unused for several minutes. While actual use Ficus in a large scale environment is so far limited to installations including only a cluster of hosts at UCLA connected to individual machines at ISI and SRI, our initial experience

[7] This reflects our philosophy that the system should never throw away data in which interest has been expressed, in this case, by updating it.

with this approach to autografting is quite positive.

References

[1] Richard G. Guy. *Ficus: A Very Large Scale Reliable Distributed File System.* Ph.D. dissertation, University of California, Los Angeles, 1991. In preparation.

[2] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63–71. USENIX, June 1990.

[3] Richard G. Guy and Gerald J. Popek. Algorithms for consistency in optimistically replicated file systems. Technical Report CSD-910006, University of California, Los Angeles, March 1991. Submitted for publication.

[4] John S. Heidemann and Gerald J. Popek. A layered approach to file system development. Technical Report CSD-910007, University of California, Los Angeles, March 1991. Submitted for publication.

[5] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[6] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Mason, Shu-Tsui Tu, and Edward R. Zayas. Decorum file system architectural overview. In *USENIX Conference Proceedings*, pages 151–163. USENIX, June 1990.

[7] Michael Leon Kazar. Synchronization and caching issues in the Andrew File System. In *USENIX Conference Proceedings*, pages 31–43. USENIX, February 1988.

[8] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Conference Proceedings*, pages 238–247. USENIX, June 1986.

[9] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *IEEE Computer*, pages 23–36, February 1988.

[10] Thomas W. Page, Jr., Richard G. Guy, Gerald J. Popek, and John S. Heidemann. Architecture of the Ficus scalable replicated file system. Technical Report CSD-910005, University of California, Los Angeles, March 1991. Submitted for publication.

[11] D. Stott Parker, Jr., Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

[12] Gerald J. Popek and Bruce J. Walker. *The Locus Distributed System Architecture.* The MIT Press, 1985.

[13] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network File System. In *USENIX Conference Proceedings*, pages 119–130. USENIX, June 1985.

[14] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[15] Edward R. Zayas and Craig F. Everhart. Design and specification of the cellular Andrew environment. Technical Report CMU-ITC-070, Carnegie-Mellon University, August 1988.

Richard Guy received a B.S. in Math/Computing from Loma Linda University in 1981, and a M.S. in Computer Science from UCLA in 1987. He was a member of the LOCUS project from 1981-87. He has been researching data replication since 1985, and is architect of the Ficus file system. He expects to receive a Ph.D. in Computer Science from UCLA in 1990.

John Heidemann received his B.S. in Computer Science at the University of Nebraska-Lincoln. He is pursuing a Ph.D. in distributed systems at UCLA. Currently his research focuses on stackable file system design. His areas of interest are distributed systems, operating systems, and file systems. He is a member of ACM and IEEE.

Wai Mak is a programmer analyst at the University of California at Los Angeles. She received her M.S. in Computer Engineering from the University of Southern California in 1989. She is a member of ACM & IEEE Computer Society. Her interest is in computer networks and distributed systems.

Tom Page received the Ph.D. and M.S degrees from UCLA in 1989 and 1983 respectively, with a major field in Distributed Operating Systems. He received a B.S. in Mathematics and Computer Science from Duke University in 1981. While a graduate student, Tom worked on the LOCUS distributed operating system and the Tangram object-oriented modeling environment. He is currently employed as a researcher at UCLA pursuing interests in distributed operating systems and distributed databases.

Gerald Popek has been a professor of Computer Science at UCLA since 1973, having received his doctorate from Harvard. He is best known for his work on the Secure Unix kernel and the design of the Locus distributed Unix system and has authored more than 50 technical papers. Popek is one of the founders of Locus Computing Corporation and is active in various industry service groups including the National Defense Service Board, the World Bank, OSF, and the Unix International Work Group on Multiprocessing.

Dieter Rothmeier is a staff member at the UCLA Computer Science Department. He received a B.S. in Mathematics/Computer Science and an M.S. in Computer Science, both from UCLA. Among his current interests are operating systems and distributed file systems.

Exploiting Multiple I/O Streams to Provide High Data-Rates

Luis-Felipe Cabrera
IBM Almaden Research Center
Computer Science Department

Internet: cabrera@ibm.com

Darrell D. E. Long
Computer & Information Sciences
University of California at Santa Cruz

Internet: darrell@sequoia.ucsc.edu

Abstract

We present an I/O architecture, called Swift, that addresses the problem of data-rate mismatches between the requirements of an application, the maximum data-rate of the storage devices, and the data-rate of the interconnection medium. The goal of Swift is to support integrated continuous multimedia in general purpose distributed systems.

In installations with a high-speed interconnection medium, Swift will provide high data-rate transfers by using multiple slower storage devices in parallel. The data-rates obtained with this approach scale well when using multiple storage devices and multiple interconnections. Swift has the flexibility to use any appropriate storage technology, including disk arrays. The ability to adapt to technological advances will allow Swift to provide for ever increasing I/O demands. To address the problem of partial failures, Swift stores data redundantly.

Using the UNIX operating system, we have constructed a simplified prototype of the Swift architecture. Using a single Ethernet-based local-area network and three servers, the prototype provides data-rates that are almost three times as fast as access to the local SCSI disk in the case of writes. When compared with NFS, the Swift prototype provides double the data-rate for reads and eight times the data-rate for writes. The data-rate of our prototype scales almost linearly in the number of servers and the number of network segments. Its performance is shown to be limited by the speed of the Ethernet-based local-area network.

We also constructed a simulation model to show how the Swift architecture can exploit storage, communication, and processor advances, and to locate the components that will limit I/O performance. In a simulated gigabit/second token ring local-area network the data-rates are seen to scale proportionally to the size of the transfer unit and to the number of storage agents.

Keywords: Swift architecture, high-performance storage systems, distributed file systems, distributed disk striping, high-speed networks, high data-rate I/O, client-server model, object server, video server, multimedia, data redundancy, resiliency.

1 Introduction

The current generation of distributed computing systems are incapable of integrating high-quality video with other data in a general purpose environment. Multimedia applications that require this level of service include scientific visualization, image processing, and recording and play-back of color video. The data-rates required by some of these applications vary

from 1.2 megabytes/second for DVI compressed video and 1.4 megabytes/second for CD-quality audio [1], to more than 20 megabytes/second for full-frame color video.

Advances in VLSI, data compression, processors, communication networks, and storage capacity mean that systems capable of integrating continuous multimedia will soon emerge. In particular, the emerging ANSI fiber channel standard will provide data-rates in excess of 1 gigabit/second over a switched network. In contrast to these advances, neither the positioning time (seek-time and rotational latency) nor the transfer rate of magnetic disks have kept pace.

The architecture we present, called Swift, solves the problem of storing and retrieving very large data objects from slow secondary storage at very high data-rates. The goal of Swift is to support integrated continuous multimedia in a general purpose distributed storage system. Swift uses disk striping [2], much like RAID [3], driving the disks in parallel to provide the required data-rate. Since Swift, unlike RAID, was designed for distributed systems it provides the advantages of easy expansion and load sharing. Swift also provides better resource utilization since it will use only those resources that are necessary to satisfy the request. In addition, Swift has the flexibility to use any appropriate storage technology, including a RAID or an array of digital audio tapes.

Two studies have been conducted to validate the Swift architecture. The first was a proof-of-concept prototype of a simplified version of Swift implemented on an Ethernet-based local-area network using the UNIX[1] operating system. This prototype provides a file system with UNIX semantics. It uses distributed disk striping over multiple servers to achieve high data-rates. On a single Ethernet-based local-area network, the prototype achieves data-rates up to three times as fast as the data-rate to access the local SCSI disk in the case of writes. When compared to a high-performance NFS file server, the Swift prototype exceeds the NFS data-rate for writes by eight times, and provides almost double the NFS data-rate for reads.

The second study is a discrete-event simulation of a simplified local-area instance of the Swift architecture. This was constructed to evaluate the effects of technological advances on the scalability of the architecture. The simulation model shows how Swift can exploit a high-speed (gigabit/second) local-area network and faster processors than those currently available, and is used to locate the components that will limit I/O performance.

The remainder of this paper is organized as follows: a brief description of the Swift architecture is described in §2 and our Ethernet-based local-area prototype in §3. Measurements of the prototype are presented in §4. Our simulation model is then presented in §5. In §6 we consider related work and present our conclusions in §7.

2 Description of the Swift Architecture

Swift builds on the idea of striping data over multiple storage devices and driving them in parallel. The principle behind our architecture is simple: aggregate arbitrarily many (slow) storage devices into a faster logical service, making all applications unaware of this aggregation. Several concurrent I/O architectures, such as Imprimis ArrayMaster [4], DataVault [5], CFS [6, 7] and RAID [3, 8], are based on this observation. Mainframes [9, 10] and super computers [11] have also exploited this approach.

[1] UNIX is a trademark of AT&T Bell Laboratories

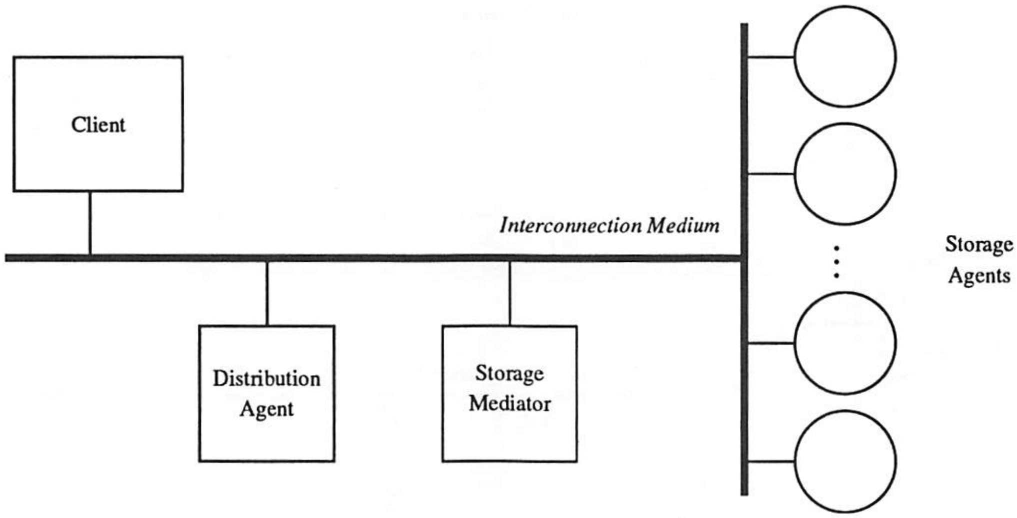


Figure 1: Components of the Swift Architecture

Swift is a distributed architecture made up of independently replaceable components. The advantage of this modular approach is that any component that limits the performance can either be *replaced* by a faster component when it becomes available or can be *replicated* and used in parallel. In this paper we concentrate on a prototype implementation and a simulation of the Swift architecture. A more detailed description of the architecture and its rationale can be found elsewhere [12, 13].

Swift assumes that objects are managed by *storage agents.* The system operates as follows: when a client issues a request to store or retrieve an object, a *storage mediator* reserves resources from all the necessary storage agents and from the communication subsystem in a session-oriented manner. The storage mediator then presents a *distribution agent* with a *transfer plan.* Swift assumes that sufficient storage and data transmission capacity will be available, and that negotiations among the client (that can behave as a *data producer* or a *data consumer*) and the storage mediator will allow the preallocation of these resources. Resource preallocation implies that storage mediators will reject any request with requirements it is unable to satisfy. To then transmit the object to or from the client, the distribution agent stores or retrieves the data at the storage agents following the transfer plan with no further intervention by the storage mediator. Figure 1 depicts the components of the Swift architecture.

In Swift, the storage mediator selects the striping unit (the amount of data allocated to each storage agent per stripe) according to the data-rate requirements of the client. If the required transfer rate is low, then the striping unit can be large and Swift can spread the data over only a few storage agents. If the required data-rate is high, then the striping unit will be chosen small enough to exploit all the parallelism needed to satisfy the request.

Partial failures are an important concern in a distributed architecture such as Swift. If no precautions are taken, then the failure of a single component, in particular a storage agent, could hinder the operation of the entire system. For example, any object which has data in a failed storage agent would become unavailable, and any object that has data being written into the failed storage agent could be damaged. The accepted solution for this problem is to use redundant data, including the *multiple copy* [14] and *computed copy*

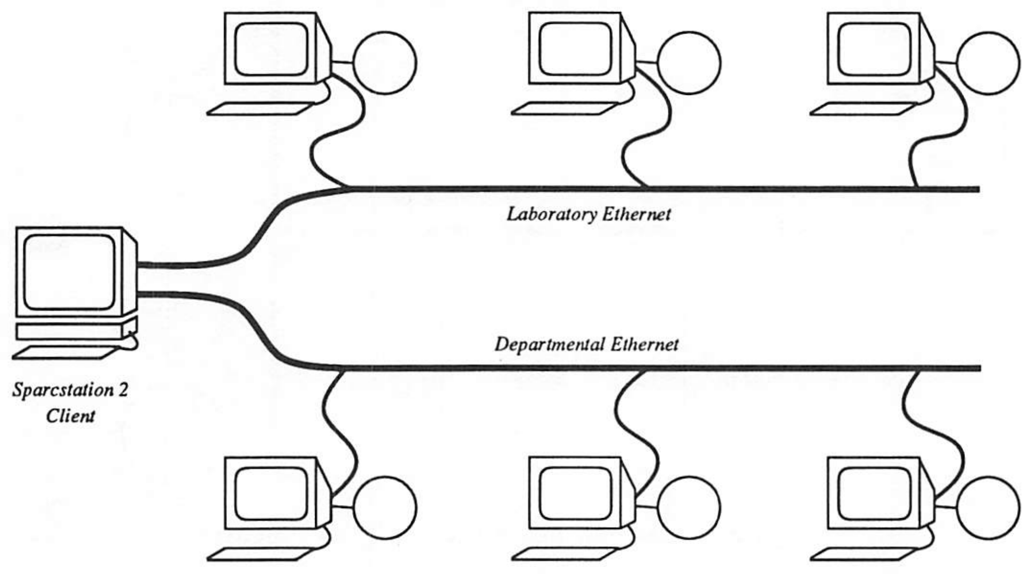


Figure 2: An Ethernet-based implementation of the Swift architecture.

[3] approaches. In the Swift prototype we propose to use computed copy redundancy since this approach provides resiliency in the presence of a single failure (per group) at a low cost in terms of storage but at the expense of some additional computation.

3 Ethernet-based Implementation of Swift

A simplified prototype of the Swift architecture has been built as a set of libraries using the standard filing and networking facilities of the UNIX operating system. We have used file system facilities to name and store objects which makes the storage mediators unnecessary. Transfer plans do not need to be built explicitly as the library interleaves data uniformly among the set of files used to service a request.

Objects administered by Swift are striped over several servers, each of which has its own local SCSI disk and act as storage agents on an Ethernet-based local-area network. Clients are provided with **open**, **close**, **read**, **write** and **seek** operations that have UNIX file system semantics.

The Swift distribution agent is embedded in the libraries and is represented by the client. The storage agents are represented by UNIX processes on servers which use the standard UNIX file system.

When an I/O request is made, the client communicates with each of the storage agents involved in the request so that they can simultaneously perform the I/O operation on the striped file. Since a file might be striped over any number of storage agents, the most important performance-limiting factors are the transmission speed of the communication network and the data-rate at which the client and the servers can send and receive packets over the network.

The current prototype has allowed us to confirm that a high aggregate data-rate can be achieved with Swift. The data-rates of an earlier prototype using a data transfer protocol

built on the TCP [15] network protocol proved to be unacceptable. The current prototype was built using a light-weight data transfer protocol on top of the UDP [15] network protocol. To avoid unnecessary data copying, scatter-gather I/O was used to have the kernel deposit the message directly into the user buffer.

In our first prototype a TCP connection was established between the client and each server. These connections were multiplexed using **select**. Since TCP delivers data in a stream with no message boundaries, a significant amount of data copying was necessary. The data-rates achieved were never more than 45% of the capacity of the Ethernet-based local-area network. At first **select** seemed to be the performance limiting factor. A closer inspection revealed that using TCP was not appropriate as buffer management problems prevented the prototype from achieving high data-rates.

In the current prototype the client is a Sun 4/75 (SPARCstation 2). It has a list of the hosts that act as storage agents. All storage agents were placed on Sun 4/20s (SLC). Both the client and the storage agents use dedicated UDP ports to transfer data and have a dedicated server process to handle the user requests.

3.1 The Data Transfer Protocol

Each Swift storage agent waits for **open** requests on a well-known IP [15] port. When an **open** request is received, a new (secondary) thread of control is established along with a private port for further communication about that file with the client. This thread remains active and the communications channel remains open until the file is **closed** by the client; the primary thread always continues to await new **open** requests.

When a secondary thread receives a **read** or **write** request it also receives additional information about the type and size of the request that is being made. Using this additional information the thread can calculate which packets are expected to be sent or received.

In the handling of a **read** operation, packet loss rates caused by lack of buffer space in the SunOS kernel necessitated that the client maintain only one outstanding packet request per storage agent, while the storage agents fulfilled the packet requests as soon as they were received. As the measurements will show, this had a negative effect on the performance of the prototype. No acknowledgments are necessary with **read**, since the client keeps sufficient state to determine what packets have been received and thus can resubmit requests when packets are lost.

With a **write** operation, the client sends out the data to be written as fast as it can. Each storage agent checks the packets it receives against the packets it was expecting and either acknowledges receipt of all packets or sends requests for packets lost. The client requires explicit acknowledgements from the storage agents to determine that a **write** operation is successful. On receipt of a **close** operation, the client expires the file handle and the storage agents release the ports and extinguish the threads dedicated to handling requests on that file.

Several problems were encountered by our current prototype with SunOS. For both **read** and **write**, we often ran out of buffer space on the client. When writing at full speed, the kernel would drop packets and claim that they had been sent. Because of this, the prototype could not write as fast as possible; we had to incorporate a small wait loop between **write** operations.

4 Measurements of the Swift Implementation

To measure the performance of the Swift prototype, three, six, and nine megabytes were read from and written to a Swift object. In order to calculate confidence intervals, eight samples of each measurement were taken. Analogous tests were performed using the local SCSI disk and the NFS file service. All data-rate measurements in this paper are given in kilobytes per second. Maintaining cold caches was achieved by using **/etc/umount** to flush the caches as a side effect. Other methods such as creating a large virtual address space to reclaim pages were also tried with similar results.

Table 1: Swift **read** and **write** data-rates on a single Ethernet in kilobytes/second.

Operation	$\bar{x}$	σ	min	max	90% Confidence	
					Low	High
Read 3 MB	893	18.6	847	904	880	905
Read 6 MB	897	3.4	891	900	894	899
Read 9 MB	876	16.6	848	892	865	887
Write 3 MB	860	44.6	767	890	830	890
Write 6 MB	882	5.00	875	889	879	885
Write 9 MB	881	1.01	857	889	874	888

The measurement data for **read** and **write** with Swift were obtained using a single client and three storage agents. The client was a Sun 4/75 (SPARCstation 2) with 16 megabytes of memory and a local 207 megabyte local SCSI disk under SunOS[2] 4.1.1. The three storage agents were Sun 4/20s with 16 megabytes of memory and a local 104 megabyte local SCSI disk also under SunOS 4.1.1. These hosts were placed on a 10 megabit/second dedicated Ethernet-based local-area network. Aside from the standard system processes, each of the servers was dedicated to run exclusively the Swift storage agent software. The results are summarized in Table 1.

Measurements of synchronous **write** operations with the Swift prototype have not been obtained at this time. We encountered a problem with SunOS that would not allow us to have the storage agents write synchronously to disk due to insufficient buffer space.

For Swift, the limiting performance factor was the Ethernet-based local-area network. Using three Swift storage agents, the utilization of the network ranged from 77% to 80% of its measured maximum capacity of 1.12 megabytes/second. Including a fourth storage agent would only saturate the network while not significantly increasing performance.

The measurements for a local SCSI disk connected to a Sun 4/20 (SLC) with 16 megabytes of memory under SunOS 4.1.1 are given in Table 2. All measurements were taken with a cold cache. The effect of the synchronous mode SCSI are most apparent for the **read** data-rates which are twice as fast as those that we obtained with version 4.1 of SunOS, which provided only asynchronous SCSI mode. All **write** operations to the SCSI disk were done synchronously.

The NFS measurements made using a Sun 4/390 with 32 megabytes of memory and

[2]The version of the operating system is significant since SunOS 4.1.1 allowed the use of synchronous mode on the SCSI drives. This doubled the read data-rate.

IPI disk drives under SunOS 4.1 as a server, and a Sun 4/75 (SPARCstation 2) as the client are summarized in Table 3. The NFS measurements were run over a lightly-loaded shared departmental Ethernet-based local-area network, not over the dedicated laboratory network. The traffic present in this shared network when the measurements were made was less than 5% of its capacity. This level of network traffic load should not significantly affect the measured data-rates.

Table 2: SCSI **read** and **write** data-rates in kilobytes/second.

Operation	$\bar{x}$	σ	min	max	90% Confidence	
					Low	High
Read 3 MB	654	10.3	641	668	647	661
Read 6 MB	671	6.4	662	682	666	674
Read 9 MB	682	2.4	679	685	680	683
Write 3 MB	314	1.3	312	316	313	315
Write 6 MB	316	0.6	315	316	315	316
Write 9 MB	315	2.1	310	316	313	316

Table 3: NFS **read** and **write** data-rates in kilobytes/second.

Operation	$\bar{x}$	σ	min	max	90% Confidence	
					Low	High
Read 3 MB	462	56.0	375	531	424	491
Read 6 MB	456	30.4	406	490	435	476
Read 9 MB	488	22.1	444	516	473	502
Write 3 MB	112	4.1	107	117	109	114
Write 6 MB	109	5.2	98	114	105	112
Write 9 MB	111	1.9	108	114	109	112

The measurements shown in Table 1 through Table 3 indicate that the Swift prototype achieves significantly higher data-rates than either the local SCSI disk or the NFS file system, even when NFS uses a high-performance server with the best IPI disk drives Sun had available[3] at the time. The Swift data-rates were up to three times better than those achieved accessing the local SCSI disk. The difference was even more accentuated a few months ago when only asynchronous SCSI mode was available. In the case of **read**, the data-rates of the Swift prototype were nearly double those of NFS, while in the case of **write** the data-rates were more than eight times those of NFS. As mentioned before, the comparison of Swift **write** data-rates with NFS **write** data-rates is not completely straightforward since NFS does synchronous writes to the disk.

When compared with the local SCSI disk performance, the Swift prototype only performs between 29% and 36% better. This contrasts sharply with previous measurements

[3]These drives were purchased Autumn 1990.

taken under SunOS 4.1 where the Swift prototype performed about 250% better than local SCSI read access. This change is attributable to the availability of synchronous SCSI mode under SunOS 4.1.1 and it supports the assertion that the performance of the Swift prototype is limited primarily by the Ethernet-based local-area network.

In contrast to its read performance, when writes are considered, the Swift prototype shows between a 274% and a 280% increase over that of the local SCSI disk. The ideal performance improvement would have been 300% if the interconnection medium were not limiting performance. Since the performance of the Swift prototype is less than 300% of the local SCSI performance, this again supports the assertion that factor most limiting the performance of the Swift prototype is the Ethernet-based local-area network.

When the Swift prototype is compared with the high-performance NFS file server, its performance is between 180% and 197% better in the case of reads. This shows that Swift can successfully provide increased I/O performance by aggregating several low-speed storage agents and driving them in parallel.

In the case of writes, the Swift prototype performs between 767% and 809% better than the high-performance NFS file server. When interpreting the measurements one should also keep in mind that the **write** data-rate measurements in NFS reflect the write-through policy of the server. This makes data-rates for **write** somewhat difficult to compare with those of Swift.

While the Swift **write** performance was measured using asynchronous writes, the values obtained are not unfair to the NFS server for two reasons. First, the local SCSI measurements were obtained using synchronous writes and are clearly more than one third the speed of the Swift performance numbers. Second, the Ethernet-based local-area network is clearly the limiting factor in the performance of the prototype. Third, the speed of the IPI disk drives (rated at more than 3 megabytes/second) should not so severely impact the performance when compared with the SCSI drives. Thus, the way in which writes are done in the Swift prototype is not the dominant performance factor.

The Swift prototype demonstrates that the Swift architecture can achieve high data-rates on a local-area network by aggregating data-rates from slower data servers. The prototype also validates the concept of distributed disk striping in a local-area network. This is demonstrated by the Swift prototype providing data-rates higher than both the local SCSI disk and the NFS file server.

4.1 Effect of Adding a Second Ethernet

To determine the effect of doubling the data-rate capacity of the interconnection, we added a second Ethernet-based local-area network segment between the client and additional storage agents. This second network segment is shared by several groups in the department. During the measurement period its load was seldom more than 5% of its capacity.

The interface for the second network sement was placed on the S-bus of the client. As the S-bus interface is known to achieve lower data-rates than the on-board interface, we did not expect to obtain data-rates twice as great as those using only the dedicated laboratory network. We also expected to see the network subsystem of the client to be highly stressed. To our surprise, our measurements show that for **write** operations the Swift prototype almost doubled its data-rate.

In the case of reads, the increase in performance of the Swift prototype is less pronounced. This can be attributed to several factors including the increased load on the

client and a lack of buffer space. It can also be attributed to the increased complexity of the read protocol which requires many more packets to be sent than does the write protocol.

The measurements for the Swift prototype using both the dedicated laboratory network and the shared departmental network are summarized in Table 4. These measurements demonstrate that the Swift architecture can make immediate use of a faster interconnection medium and that its data-rates scale accordingly.

Table 4: Swift **read** and **write** data-rates on two Ethernets in kilobytes/second.

Operation	$\bar{x}$	σ	min	max	90% Confidence	
					Low	High
Read 3 MB	1120	36.8	1040	1150	1093	1143
Read 6 MB	1150	8.5	1140	1170	1145	1156
Read 9 MB	1130	11.0	1120	1150	1126	1140
Write 3 MB	1660	10.1	1640	1670	1650	1663
Write 6 MB	1670	3.0	1660	1670	1665	1669
Write 9 MB	1660	14.3	1630	1680	1652	1671

5 Simulation-based Performance Study

To evaluate the scaling properties of the architecture we modeled a hypothetical implementation on a high-speed local-area token-ring network. The main goal of the simulation was to show how the architecture could exploit network and processor advances. A second goal was to demonstrate that distributed disk striping is a viable technique that can provide the data-rates required by applications such as multimedia.

Since we did not have the necessary network technology available to us, a simulation was the appropriate exploration vehicle. The token-ring local-area network was assumed to have a transfer rate of 1 gigabit/second. The clients were modeled as diskless hosts with a 100 million instructions/second processor and a single network interface connected to the token-ring. The storage agents were modeled as hosts with a 100 million instructions/second processor and a single disk device. In our simulation runs no more than 22% of the network capacity was ever used, and so our data-rates were never limited by lack of network capacity.

5.1 Structure of the Simulator

The simulator was used to locate the components that were the limiting factors for a given level of performance. The simulator did not model caching, did not model computing data parity blocks, did not model any preallocation of resources, nor did it attempt to provide performance guarantees. Traces of file system activity would have been required in order to model these effectively and such traces were unavailable to us. In addition, the simulator did not model the storage mediator as it is not in the path of the data transmitted to and from clients, but is consulted only at the start of an I/O session.

Components of the system are modeled by client requests and storage agent processes. A generator process creates client requests using an exponential distribution for request

interarrival times. The client requests are differentiated according to a read-to-write ratio. In each of the following figures, this ratio has been conservatively estimated to be 4:1, motivated by the Berkeley study [16] and by our belief that for continuous media read access will strongly dominate write access.

In our simulation of Swift, to **read**, a small request packet is multicast to the storage agents. The client then waits for the data to be transmitted by the storage agents. A **write** request transmits the data to each of the storage agents. Once the blocks have been transmitted the client awaits an acknowledgement from the storage agents that the data have been written to disk.

The disk devices are modeled as a shared resource. Multiblock requests are allowed to complete before the resource is relinquished. The time to transfer a block consists of the seek time, the rotational delay and the time to transfer the data from disk. The seek time and rotational latency are assumed to be independent uniform random variables, a pessimistic assumption when advanced layout policies are used [17]. Once a block has been read from disk it is scheduled for transmission over the network.

Our model of the disk access time is conservative in that advanced layout policies are not considered, no attempt was made to order requests to schedule the disk arm, and caches were not modeled. Staging data in the cache and sequential preallocation of storage would greatly reduce the number of seeks and significantly improve performance. As it is, our model provides a lower bound on the data-rates that could be achieved.

Transmitting a message on the network requires protocol processing, time to acquire the token, and transmission time. The protocol cost for all packets has been estimated at 1,500 instructions [18] plus one instruction per byte in the packet. The time to transmit the packet is based on the network transfer rate.

5.2 Simulation Results

The simulator gave us the ability to determine what data-rates were possible given a configuration of processors, interconnection medium and storage devices. The modeling parameters varied were the type and number of disk devices, representing storage agents, and the size of the transfer unit.

The clear conclusion is that when sufficient interconnection capacity is available the data-rate is almost linearly related to both the number of storage agents and to the size of the transfer unit. The reason the transfer unit impacts so much the data-rates achieved by the system is that seek time and rotational latency are enormous when compared to the speed of the processors and the network transfer rate. This also shows the value of careful data placement and indicates that resource preallocation may be very beneficial to performance.

In Figure 3, the amount of time required to satisfy a 1-megabyte client request was plotted against the number of requests issued per second. The base storage device considered was a Fujitsu M2372K, which is typical for 1990 file servers.

The load that could be carried depended both on the number of disks used and the block size. The delay was dominated by the disk, with an average seek time of 16 milliseconds, an average rotational delay of 8.3 milliseconds and a transfer rate of 2.5 megabytes per second. The result was that transferring 32 kilobytes required about 37 milliseconds on the average. As the block size was increased, seek time and rotational delay were mitigated and the transfer time became more dependent on the amount of data transferred.

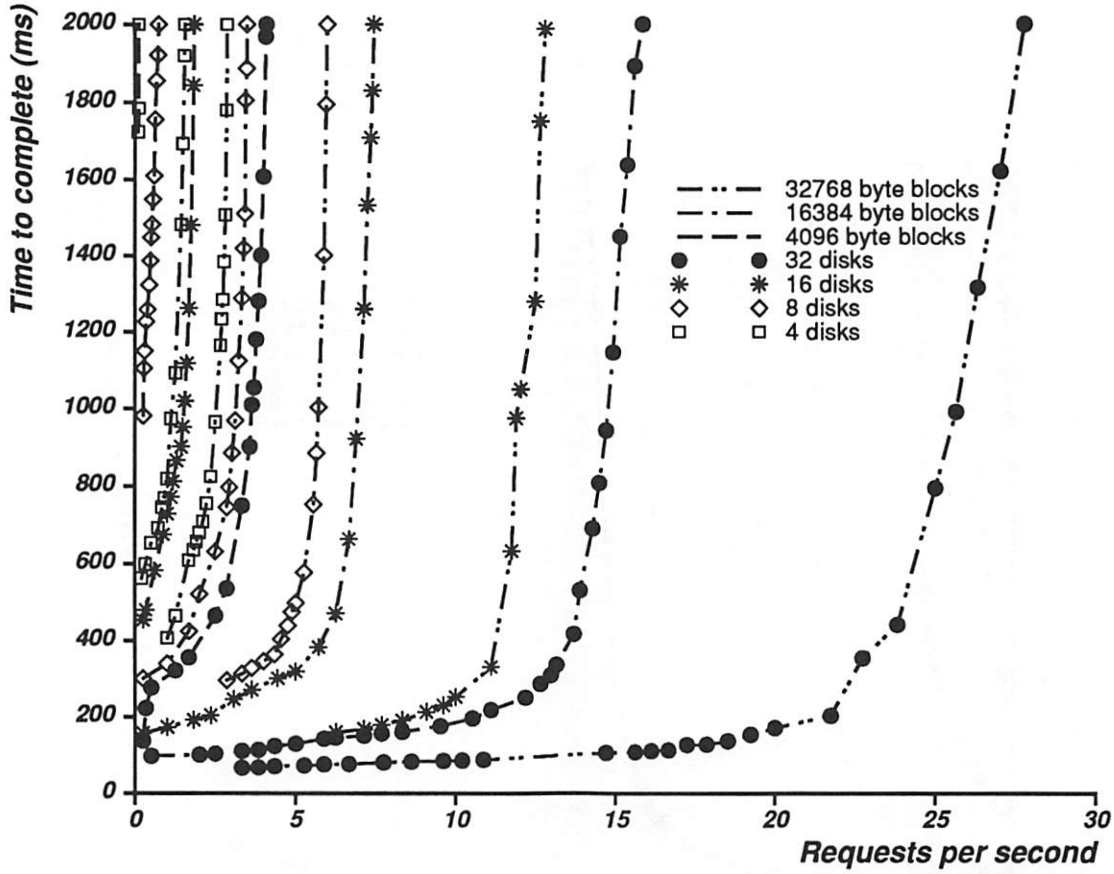


Figure 3: Average time to complete a client request.

Simulation parameters: average seek time = 16 ms, average rotational delay = 8.3 ms, transfer rate = 2.5 megabytes/second, client request = 1 megabyte, disk transfer unit = {4, 16, 32} kilobytes.

As small transfer sizes require many seeks in order to transfer the data, large transfer sizes have a significantly positive effect on the data-rates achieved. For small numbers of disks, seek time dominated to the extent that its effect on performance was almost as significant as the number of disks.

When 4 disks were used, the system saturated quickly. This is because fewer disks had to service a larger number of requests. For larger numbers of disks, the response time was almost constant until the knee in the performance curve was reached. For 32 disks, the maximum sustainable load was reached at about 22 requests per second. At this point the disks were 50% utilized on the average. The rate of requests that are serviceable increased almost linearly in the number of disks. Increased rotational delay and a slight loading of the communication medium prevents it from being strictly linear.

The data transfer processing costs also need to be taken into account. For example, it was assumed that protocol processing required 1500 instructions plus 1 instruction per byte in the packet. As the size of the packet increases, the protocol cost decreases proportionally to the packet size. The cost of 1 instruction per byte in the packet is for the most part

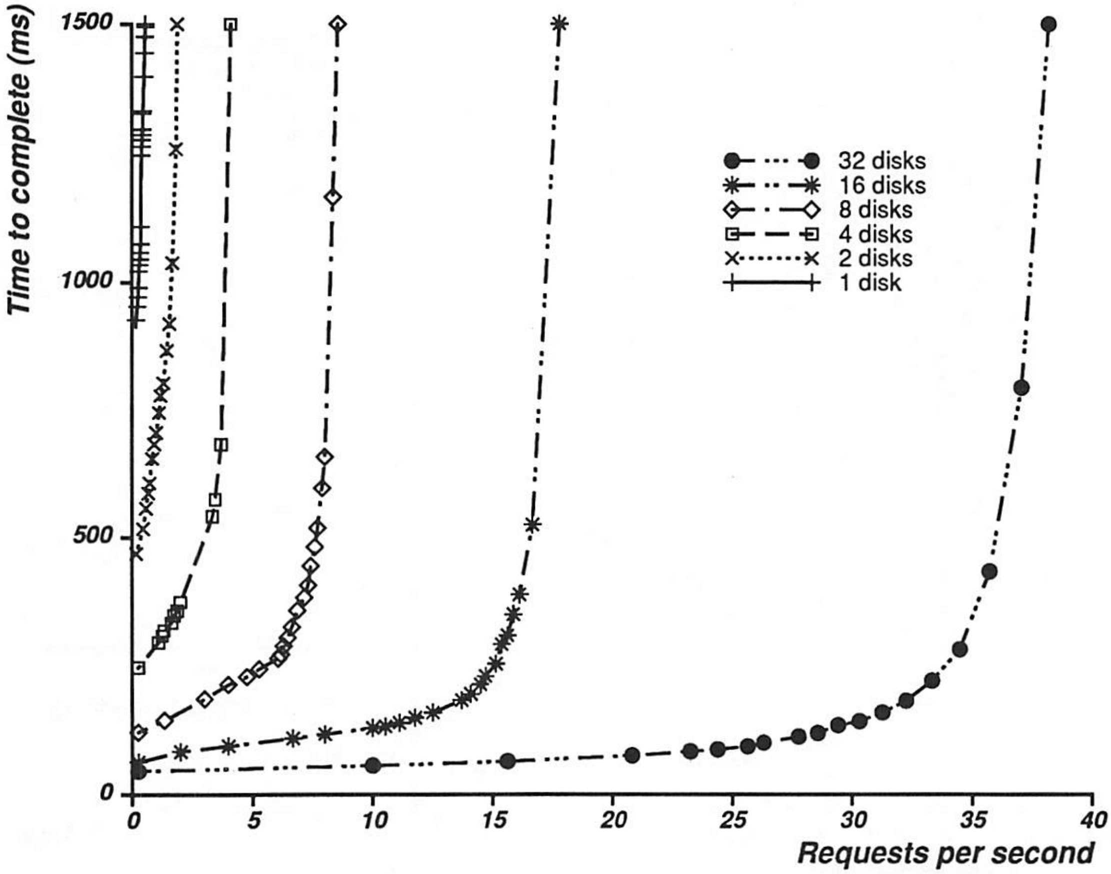


Figure 4: Average time to complete a client request.

Simulation parameters: average seek time = 16 ms, average rotational delay = 8.3 ms, transfer rate = 1.5 megabytes/second, client request = 128 kilobytes, disk transfer unit = 4 kilobytes.

unavoidable, since it is necessary data copying.

In Figure 4 we present the amount of time required to satisfy a 128 kilobyte client request using a slower storage device.

In Figures 5 and 6, the maximum sustainable data-rate is considered. The maximum sustainable data-rate is the data-rate observed by the client when the average time to complete a request is the same as the average time between requests.

The effect of seek time is seen once again. For transfer units of 4 kilobytes, the maximum sustainable data-rate for 32 disks is approximately 2 megabytes per second. When transfer units of 32 kilobytes are used, the maximum sustainable data-rate increases to nearly 12 megabytes per second for the same 32 disks. The increase in effective data-rate is almost linear in the size of the transfer unit.

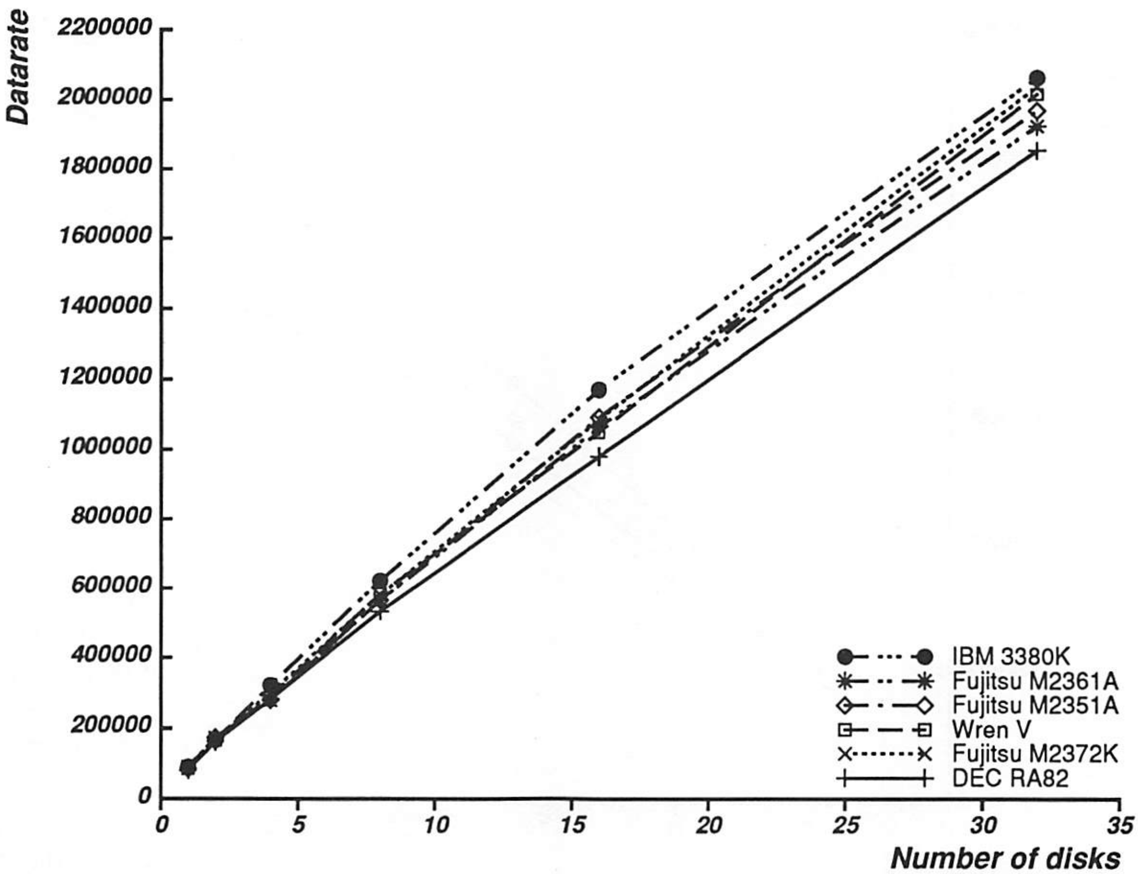


Figure 5: Observed client data-rate at maximum sustainable load.

Simulation parameters: client request = 128 kilobytes, disk transfer unit = 4 kilobytes.

6 Related Research

The notion of *disk striping* was formally introduced by Salem and Garcia-Molina [2]. The technique, however, has been in use for many years in the I/O subsystems of super computers [11] and high-performance mainframe systems [9]. Disk striping has also been used in some versions of the UNIX operating system as a means of improving swapping performance [2]. To our knowledge, Swift is the first to use disk striping in a distributed environment, striping files over multiple servers.

Examples of some commercial systems that utilize disk striping include super computers [11], DataVault for the CM-2 [5], the airline reservation system TPF [9], the IBM AS/400 [10], CFS from Intel [6, 7], and the Imprimis ArrayMaster [4]. Hewlett-Packard is developing a system called DataMesh that uses an array of storage processors connected by a high-speed switched network [19]. For all of these the maximum data-rate is limited by the interconnection medium which is an I/O channel. Higher data-rates can be achieved by using multiple I/O channels.

The aggregation of data-rates proposed in the Swift architecture generalizes that proposed by the RAID disk array system [3, 8, 20] in its ability to support data-rates beyond

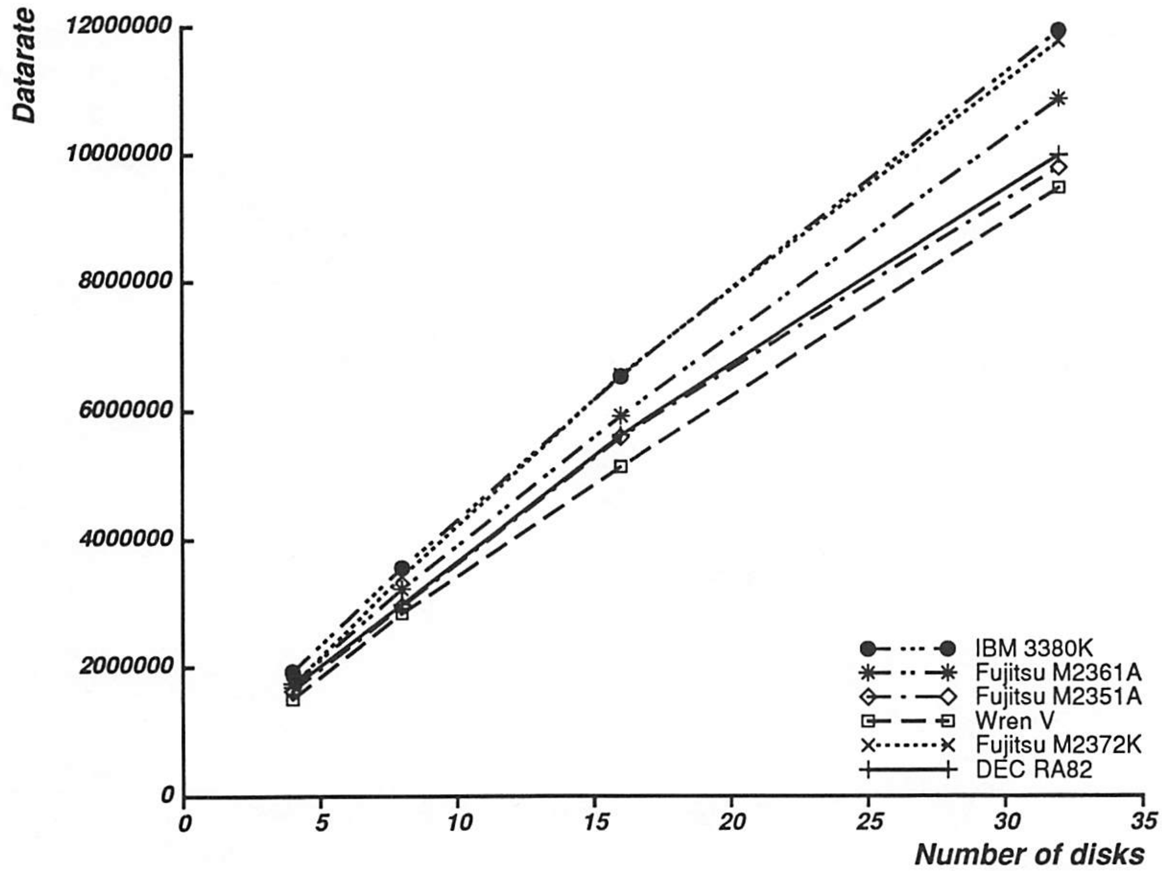


Figure 6: Observed client data-rate at maximum sustainable load.

Simulation parameters: client request = 1 megabyte, disk transfer unit = 32 kilobytes.

that of the single disk array controller. In fact, Swift can concurrently drive a collection of RAIDs as high speed devices. Due to the distributed nature of Swift, it has the further advantage over RAID of having no single point of failure, such as the disk array controller or the power supply.

Swift differs from traditional disk striping systems in two important areas: scaling and reliability. By interconnecting several communication networks Swift is more scalable than centralized systems. When higher performance is required additional storage agents can be added to the Swift system increasing its performance proportionally. By selectively hardening each of the system components, Swift can achieve arbitrarily high reliability of its data, metadata, and communication media. In CFS, for example, there is no mechanism present to make its metadata tolerant of storage failures. In CFS if the repository on which the descriptor of a multi-repository object fails, the entire object becomes unavailable.

A third difference from traditional disk striping systems is that Swift has the advantages of sharing and of decentralized control of a distributed environment. Several independent storage mediators may control a common set of storage agents. The distributed nature of Swift allows better resource allocation and sharing than a centralized system. Only those resources that are required to satisfy the request need to be allocated.

Swift incorporates data management techniques long present in centralized computing systems into a distributed environment. In particular, it can be viewed as a generalization to distributed systems of I/O channel architectures found in mainframe computers [21].

6.1 Future Work

There are two areas that we intend to address in the future: enhancing our current prototype and simulator, and extending the architecture.

6.1.1 Enhancements to the Prototype and the Simulator

Both the current prototype and the simulator need to address data redundancy. The prototype will be enhanced with code that computes the check data, and both the **read** and **write** operations will have to be modified accordingly. The simulator needs additional parameters to incorporate the cost of computing this derived data. With these enhancements in place we plan to study the impact that computing the check data has on data-rates.

We also plan to incorporate mechanisms to do resource preallocation and to build transfer plans. With these mechanisms in place we plan to study different resource allocation policies, with the goal of understanding how to handle variable loads.

6.1.2 Enhancements to the Architecture

We intend to extend the architecture with techniques for providing data-rate guarantees for magnetic disk devices. While the problem of real-time processor scheduling has been extensively studied, and the problem of providing guaranteed communication capacity is also an area of active research, the problem of scheduling real-time disk transfers has received considerably less attention.

A second area of extensions is in the co-scheduling of services. In the past, only analog storage and transmission techniques have been able to meet the stringent demands of multimedia audio and video applications. To support integrated continuous multimedia, resources such as the central processor, peripheral processors (audio, video), and communication network capacity must be allocated and scheduled together to provide the necessary data-rate guarantees.

7 Conclusions

This paper presents two studies conducted to validate Swift, a scalable distributed I/O architecture that achieves high data-rates by striping data across several storage devices and driving them concurrently. The prototype validates the concept of distributed disk striping in a local-area network.

A prototype of Swift was built using UNIX and an Ethernet-based local-area network. It demonstrated that the Swift architecture can achieve high data-rates on a local-area network by aggregating data-rates from slower data servers. The prototype achieved up to three times faster data-rates than the data-rate to access the local SCSI disk, and it achieved eight times the NFS data-rate for writes and almost twice the NFS data-rate for reads. The performance of our local-area network Swift prototype was limited by the speed of the Ethernet-based local-area network.

When a second Ethernet path was added between the client and the storage agents, the data-rates measured demonstrated that the Swift architecture can make immediate use of a faster interconnection medium. The data-rates for **write** almost doubled. For **read**, the improvements were only on the order of 25% because the client could not absorbe the increased network load.

Second, simulations show how Swift can exploit more powerful components in the future, and where components limiting I/O performance will be. The simulations show that data-rates under Swift scale proportionally to the size of the transfer unit and the number of storage agents when sufficient interconnection capacity is available.

Even though Swift was designed with very large objects in mind, it can also handle small objects, such as those encountered in normal file systems. The penalties incurred are one round trip time for a short network message, and the cost of computing the parity code. Swift is also well suited as a swapping device for high performance work stations if no data redundancy is used.

The distributed nature of Swift leads us to believe that it will be able to exploit all the current hardware trends well into the future: increases in processor speed and network capacity, decreases in volatile memory cost, and secondary storage becoming very inexpensive but not much faster. The Swift architecture also has the flexibility to use alternative data storage technologies, such as arrays of digital audio tapes.

Lastly, a system like our prototype can be installed easily using much of the existing hardware. It can be used to fully exploit the emerging high-speed networks and the large installed base of file servers.

Acknowledgements. We are grateful to those that contributed to this work including Daniel Edelson for his assistance with the simulator, Aaron Emigh for his work on the prototype and in measuring its performance, Laura Haas, Richard Golding and Mary Long for their thoughtful comments on the manuscript, Dean Long for building the custom kernels necessary for our measurements, and John Wilkes for stimulating discussions on distributed file systems. Darrell Long was supported in part by a University of California Regents Junior Faculty Fellowship.

References

[1] A. C. Luther, *Digital Video in the PC Environment.* McGraw-Hill, 1989.

[2] K. Salem and H. Garcia-Molina, "Disk striping," in *Proceeding of the* 2nd *International Conference on Data Engineering*, pp. 336–342, IEEE, Feb. 1986.

[3] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the ACM SIGMOD Conference*, (Chicago), pp. 109–116, ACM, June 1988.

[4] Imprimis Technology, *ArrayMaster 9058 Controller*, 1989.

[5] Thinking Machines, Incorporated, *Connection Machine Model CM-2 Technical Summary*, May 1989.

[6] P. Pierce, "A concurrent file system for a highly parallel mass storage subsystem," in *Proceedings of the* 4th *Conference on Hypercubes*, (Monterey), Mar. 1989.

[7] T. W. Pratt, J. C. French, P. M. Dickens, and S. A. Janet, "A comparison of the architecture and performance of two parallel file systems," in *Proceedings of the* 4$^{\text{th}}$ *Conference on Hypercubes*, (Monterey), Mar. 1989.

[8] S. Ng, "Pitfalls in designing disk arrays," in *Proceedings of the IEEE COMPCON Conference*, (San Francisco), Feb. 1989.

[9] IBM Corporation, *TPF-3 Concepts and Structure Manual.*

[10] B. E. Clark and M. J. Corrigan, "Application System/400 performance characteristics," *IBM Systems Journal*, vol. 28, no. 3, pp. 407–423, 1989.

[11] O. G. Johnson, "Three-dimensional wave equation computations on vector computers," *Proceedings of the IEEE*, vol. 72, Jan. 1984.

[12] L.-F. Cabrera and D. D. E. Long, "Swift: A storage architecture for large objects," in *Proceedings of the* 11$^{\text{th}}$ *Symposium on Mass Storage Systems*, (Monterey, California), IEEE, Oct. 1991.

[13] L.-F. Cabrera and D. D. E. Long, "Swift: A storage architecture for large objects," Tech. Rep. IBM Almaden Research Center RJ7128, International Business Machines, Oct. 1990.

[14] S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in partitioned networks," *Computing Surveys*, vol. 17, pp. 341–370, Sept. 1985.

[15] D. E. Comer, *Internetworking with TCP/IP: Principles, Protocols, and Architecture.* Prentice-Hall, 1988.

[16] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, "A trace-driven analysis of the UNIX 4.2 BSD file system," in *Proceedings of the* 10$^{\text{th}}$ *Symposium on Operating System Principles*, (Orcas Island, Washington), pp. 15–24, ACM, Dec. 1985.

[17] F. Douglis and J. Ousterhout, "Log-structured file systems," in *Proceedings of the IEEE COMPCON Conference*, (San Francisco), Feb. 1989.

[18] L.-F. Cabrera, E. Hunter, M. J. Karels, and D. A. Mosher, "User-process communication performance in networks of computers," *IEEE Transactions on Software Engineering*, vol. 14, pp. 38–53, Jan. 1988.

[19] J. Wilkes, "DataMesh — project definition document," Tech. Rep. HPL-CSP-90-1, Hewlett-Packard Laboratories, Feb. 1990.

[20] S. Ng, "Some design issues of disk arrays," in *Proceedings of the IEEE COMPCON Conference*, (San Francisco), Feb. 1989.

[21] J. Buzen and A. Shum, "I/O architecture in MVS/370 and MVS/XA," *ICMG Transactions*, vol. 54, pp. 19–26, 1986.

Luis-Felipe Cabrera received the B.S. degree and M.S. degree (Mathematics) from the Pontificia Universidad Catolica de Chile, Santiago in 1974 and 1975 respectively, and the M.S. degree (Computer Science) and Ph.D. degree from the University of California at Berkeley, Berkeley, CA in 1979 and 1981 respectively.

From 1981-1983 he was a faculty in the Departamento de Ciencia de la Computacion of the Pontificia Universidad Catolica de Chile, being its Chairman during 1982-1983. In January 1984 he joined the faculty of the Computer Science Division of the Electrical Engineering and Computer Sciences Department of the University of California at Berkeley. While at Berkeley his research focused on Performance Evaluation and Distributed Systems. During the summers of 1984 and 1985 he was the Research Supervisor of Berkeley's Computer Systems Research Group, CSRG, the makers of Berkeley UNIX. He was also a consultant for Xerox PARC and Hewlett Packard.

In late 1985 he joined Computer Science Department of IBM Almaden Research Center. At Almaden he was the co-designer and co-builder of a transactional file service, QuickSilver DFS. He then helped start the research effort Melampus, that explores the limit when the operating system and the data base management system cease to be different and provide a common framework to administrate all entities in a computing system. His current research interests include distributed computing systems, object-oriented systems, high-performance storage repositories, performance evaluation, and software performance.

Dr. Cabrera has published some 40 research articles in journals and conferences, including IEEE Transactions on Software Engineering, USENIX, IEEE International Conference on Distributed Computing Systems, IEEE Conference on Workstations, IEEE COMPSAC, and CPEUG.

Dr. Cabrera may be reached at IBM Almaden Research Center, K55/803, Computer Science Department, San Jose, CA 95120-6099 or via email: cabrera@ibm.com.

Darrell D. E. Long received his B.S. degree in Computer Science from San Diego State University in 1984. He received his M.S. degree in 1986 and his Ph.D. degree in 1988, both in Computer Science from the University of California, San Diego. He is now Assistant Professor of Computer and Information Sciences at the University of California at Santa Cruz.

Dr. Long has published research articles on protocols for data replication, wide-area replication, reliability and high-speed I/O systems. He is also the author (with John Carroll) of a text on automata theory and formal languages published by Prentice-Hall. His current research interests include distributed computing systems, high-speed I/O systems, performance evaluation and computer system security.

Dr. Long is a member of the IEEE Computer Society, the Association for Computing Machinery and Sigma Xi. He is an associate editor of the International Journal of Computer Simulation, and editor of the IEEE Technical Commitee on Operating Systems and Application Environments newsletter.

Dr. Long may be reached at Computer & Information Sciences, University of California Santa Cruz, CA 95064 or via email: darrell@cis.ucsc.edu.

An Open and Extensible Event-based Transaction Manager

USENIX Summer 1991 Conference

Edward C. Cheng
Edward Chang
Johannes Klein
Dora Lee
Edward Lu
Alberto Lutgardo
Ron Obermarck

TP West
Digital Equipment Corporation
cheng@tpwest.dec.com

Abstract: The concept of a transaction has been used to ensure atomicity, consistency, isolation, and durability of a unit of work in a computing environment. A number of transaction models have been proposed and defined over the years to relate multiple units of works together. Two distinguishing properties are observed among the various transaction models: namely, concurrency control and recovery. Depending upon the type of application, it may be desirable to maintain different concurrent access rules as well as variant recovery coordination schemes over a transaction tree that is composed of a number of transaction models. In the past, there have been attempts to implement a transaction manager as a system service to provide transactional coordination on both the system and user level. However, these traditional approaches can support only a limited number of transaction models due to the complexity of coping with the various and sometimes contradictory transaction properties. Indeed, many of these transaction managers assume only the flat transaction model which clearly will not be sufficient in today's open computing environment.

In this paper, we present an approach to an open and extensible transaction manager that uses an event triggering and synchronization mechanism. Transaction models are defined based on transaction dependency rules to relate transaction instances together [8]. The transaction manager pre-defines a set of primitive relations between transaction agents; from this pool of dependencies, a user can choose a combination of rules as desired policies to govern the behavior of transactions. By using primitive dependencies, multiple transaction models can be constructed within the same transaction tree and are able to coexist with one another. Interoperability between different transaction managers is also made possible with the emulation of different transaction models by using the transaction-related dependencies.

1 Introduction

The concept of a transaction has been developed to permit the management of jobs and resources in a

reliable computing environment [5]. Up until now, the work of transactions has been focused primarily on the flat transaction model in which if one component of a global job fails, the entire computation will have to be undone. This is obviously inefficient and undesirable in today's complex application environment where multiple processes within a relatively long-lived transaction may span across serveral workstations or PC's; when one of these processes or nodes fails, users would like to retry only that portion of work on the same node or on a different node. Also when one component has completed its work, in some cases, it is more efficient to release its resources for other processes to use instead of holding locks on those resources until the global transaction commits. Users simply cannot afford the high cost of such a stringent commit-abort and concurrency control policy as imposed by the flat transaction model. We are convinced that supporting solely the simple flat transaction model will not satisfy the computer users of the 90's.

Unfortunately, to develop a transaction manager which can support multiple transaction models is not an easy task. In the past, a transaction management system has been developed both on the resource manager level as well as the kernel level to coordinate the states of multiple transactions [7]. Traditionally, a designer took the approach of implementing such transaction managers by hard coding the desired transaction-related properties as mapped to a particular transaction model. As a result, it is very difficult to extend the design to capture more properties that are involved in different transaction models. The responsibility of coordinating transactional activities within an application which uses multiple transaction models is therefore laid upon application logic. This coordination becomes a non-trivial issue for application developers. Furthermore, in an open environment, one also finds it difficult to allow multiple transaction managers to interoperate with one another simply because of the fact that they may have different semantic meaning for transaction events and properties.

In this paper, we propose an approach of building a transaction manager by using some low-level transaction dependencies [9]. Through the use of such primitive dependencies, we succeed in describing the various known transaction models, and, more significantly, we are able to coordinate the state transition among transaction instances that are bound together by the different transaction models. In the next two sections we describe the conceptual view of transaction models, agents, and events. Following that, the dependencies and their usage to construct transaction models will be discussed. An event-based transaction manager which utilizes an event triggering and synchronization mechanism to maintain such transaction dependencies is covered in section 5. In section 6 we will discuss some applications which use mixed transaction models.

2 Transaction Models

A distributed complex computation usually involves a number of subcomputations spread over a heterogenous computer network. Each of these subcomputations or computation components may employ the idea of a transaction to ensure its own atomicity and consistency in a multi-user environment. In order to guarantee certain transactional behavior over the whole distributed computation, the transaction components are linked together to form a global transaction or a transaction tree. A defined set of behavior is maintained between any two subtransactions by a transaction manager. Each distinctive set of transactional behavior is termed a transaction model. These concepts can be depicted by the following expression:

Transaction Component, t_i	=	a unit of local computation or non-undoable action
Transaction Model, M_{ij}	=	$\{ b_1, b_2, ..., b_n \}$ where b_k is a transactional behavior between transaction components t_i and t_j
Transaction, T	=	$\{ t_i, t_j, t_k, ..., M_{ij}, M_{ik}, ... \}$

In the past, one transaction model was used within a transaction T that was maintained by a single trans-

action manager. Much work has been done to define the various transaction models. These models include simple, flat, nested, multi-level, chained, multi-nested and others [1,2,3,4,6]. All the models are specified by a set of behavior which relates to rollback recovery and concurrent data sharing. We will further describe transaction models in section 4 after we have introduced the concept of agents and dependencies.

3 Agents and Events

In a computing environment, an agent is associated with a component which is responsible for carrying out certain computations or actions as defined by an application[9,13]. An agent therefore may be affiliated with a high-level application program; such an agent is termed an AP-agent. Alternately, an agent may be associated with a resource manager, and is called an RM-agent. Each agent has its own function as defined by the associated program [13]. At any one point in time, an agent has a current state, a context, and a set of dependencies. An agent may move from one state to another upon receiving an event. In the view of transaction processing, an agent lives through a deterministic set of transaction-related events, has a context that is maintained by itself, and has a set of dependencies which relates this agent to the outside world. In the model of event-based transaction management, the states of an agent are internal to itself; the agent communicates its state transitions with others by sending events.

Events

Agents go through a sequence of events. Note that the transaction manager, which is responsible for coordinating activies among the transaction agents, is only required to ensure that an agent enters from a beginning event and exits with a termination event. The intermediate events are exposed only if the dependency between this agent and other agents imposes such a necessity. Events are intercepted by the transaction manager only if such events are required to be synchronized with other events from other agents. An agent may move from one event to another upon receiving an *external* event from another agent. An agent may also go through such a transition by triggering an *internal* event to itself. The events defined in our transaction manager are as follows:

Create An event sent by a client to request the creation of a transaction agent instance. This event maps to the creation of the agent block in main memory and the associated dependencies specified by the request.

Finish An event to notify other agents that the sender has finished its work and that no more data access or update will be done after this point. This event is only sent to the agents who are *finish dependent* on the sender. In other words, the finish event is externalized only when there is a *finish dependency*.

Req_prepare In the case where mutual *commit dependencies* exist, the two agents (or a number of agents) have to coordinate their commitment by making promises. Req_prepare requests a promise that an agent can go commit its work if desired.

Prepare A promise event to indicate that an agent agrees and is able to commit.

Commit The commit event is sent to indicate that an agent has arrived at the committing state. Note that this event is not necessarily related to the handshaking situation as described under Req-prepare. But it should be viewed as an event to expose the commit state of an agent.

Forget The forget event is sent by the dependent agent when the commit dependency is resolved and the agent it depends on can be forgotten.

Abort The abort event is sent to abort an agent.

The event sequence can be depicted by the following figure:

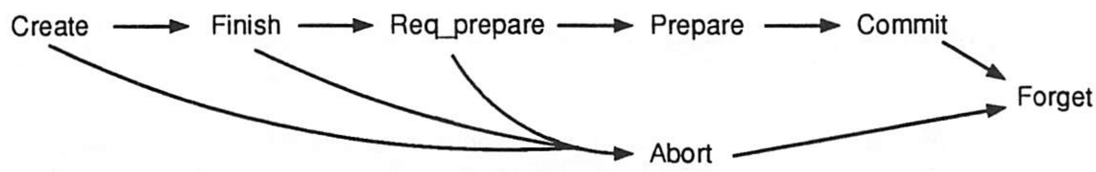


Figure 1 Event Sequence of Transaction Agents

4 Dependencies

An agent relates to the outside world by declaring dependencies with other agents. A dependency is defined by *relating* two events together. Two types of relations are defined in the transaction manager: the *ordering relationship* and the *implication relationship.* The ordering relationship between two events of two agents indicates the order in which the agents may trigger certain events; it has to do with the relative time-ordering of the agents' activities. The implication relationship between two events defines the requirement that an agent may trigger an event if and only if the other agent either has triggered an event or promises to trigger an event. The promise here has to stand in accordance with the transaction protocol as well as against a system or communication failure. Different transaction dependencies can be derived from these two types of relations. The following transaction dependencies are used to describe the transaction models:

Strong Commit Dependency $\mathrm{Commit}_{A1} \longrightarrow \mathrm{Commit}_{A2}$

Agent A2 is strong commit dependent on agent A1 if the commitment of A2 implies the commitment of A1.

Weak Commit Dependency $\mathrm{Commit}_{A1} \dashrightarrow \mathrm{Commit}_{A2}$

Agent A2 is weak commit dependent on agent A1 if the commitment of A2 implies either the commitment of A1 if it has finished or otherwise the abortion of A1. Note that the weak commit dependency will become a strong commit dependency when the agent (A1) on which there is a commit dependency by another agent (A2) has finished. In other words, A2's weak commit dependency on A1 will become a strong commit dependency when A1 finishes. The rationale behind this is that A2 may have seen the result of A1 after A1 finished.

Finish Dependency $\mathrm{Finish}_{A1} \longrightarrow \mathrm{Finish}_{A2}$

Agent A2 is finish dependent on agent A1 if A2 can finish only after A1 has finished

The finish dependency is an ordering relationship, while the commit dependencies are within the class of an implication relationship.

We can now construct different transaction models by using these three dependencies. A flat transaction

can be represented by mutual commit dependency. Figure 2 shows the agents and the dependency arcs.

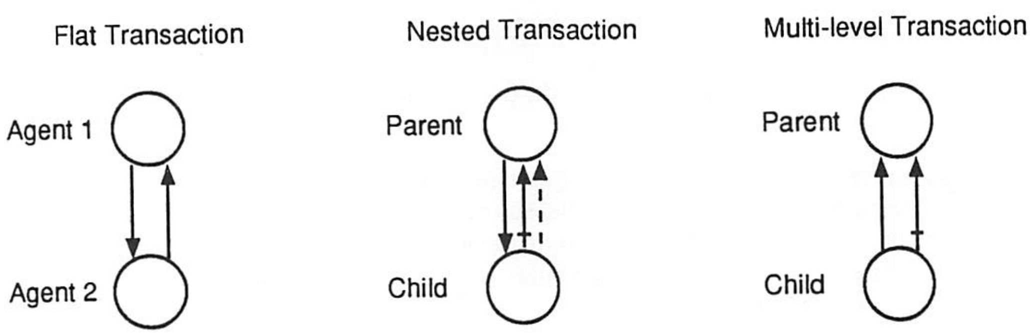


Figure 2 Transaction Models Constructed by Using Dependencies

When one agent aborts, the other agent also has to abort. On the other hand, for one agent to commit, the other agent must have either committed or promised to commit. Similarly, a nested transaction can be composed by using all three dependencies. As shown in Figure 1, before a child has finished, the abort of the child will not impact the commitment of the parent. Note that at any point in time if the parent aborts, the child will abort. The transaction manager, upon receiving an event indicating the abort of the parent, will trigger an abort event to the child to accomplish such an effect. This triggering and synchronization of events will be covered in the next section. In the case of multi-level transaction, only the parent is dependent on the child. The child therefore can commit anytime it desires. However, failure of the child will cause the parent transaction to abort. In the case where the parent has aborted after the child has committed, one may want to compensate for the committed effects that have been done by the child. A compensation agent can be defined in this model by using dependencies such that the compensated transaction agent is *abort-commit* depend on the parent and the child [9]. In other words, a compensation agent will only commit if the parent aborts and the child commits.

The role of the transaction manager in this model is to synchronize external events from the agents according to the dependencies that have been defined by the applications and the resource managers.

5 Event-based Transaction Manager

Transaction management (TM) systems are used to coordinate transaction activities on a system [10,11,12]. In the event-based transaction manager (ETM) described here, the user has the option to specify the relationship between this transaction component and another at the time the transaction starts. From the point of view of the ETM, the transaction tree is made up of agents and dependency arcs (Figure 3). It synchronizes activities according to the dependencies instead of the transaction models. In fact,

transaction models do not have any special meaning to ETM. By using primitive dependencies, multiple

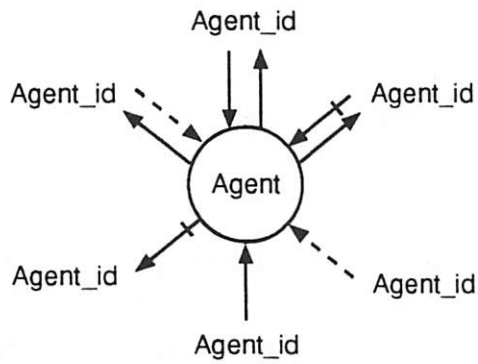


Figure 3 An Agent and Its Dependencies in the ETM

transaction relations can be combined together to generate a virtually infinite number of transaction models which can be maintained by a single ETM. When an event is delivered to an agent, the ETM looks up the conditions implied by the agent's dependencies to decide if one or multiple events shall be triggered to other agents. In the following example, an application creates two subtransactions and relates them by a mutual strong commit dependency:

```
Begin_Transaction(Agent_id1, NULL, NULL);              /* begin the top tranx    */
Begin_Transaction(Agent_id2, Agent_id1, SCD|NSCD);     /* begin Agent 2 in Flat  */
```

On the first Begin_Transaction call, a create event is sent to the transaction manager and Agent 1 is created. The data structure which reflects this agent is kept in the transaction manager. When the second request is submitted, Agent 2 is created along with a mutual strong commit dependency (SCD | NSCD, where NSCD stands for the SCD in an opposite direction) relation with Agent 1. The data blocks of both agents and their dependency arcs are maintained by the transaction manager. One may look at the relationship as shown in Figure 4.

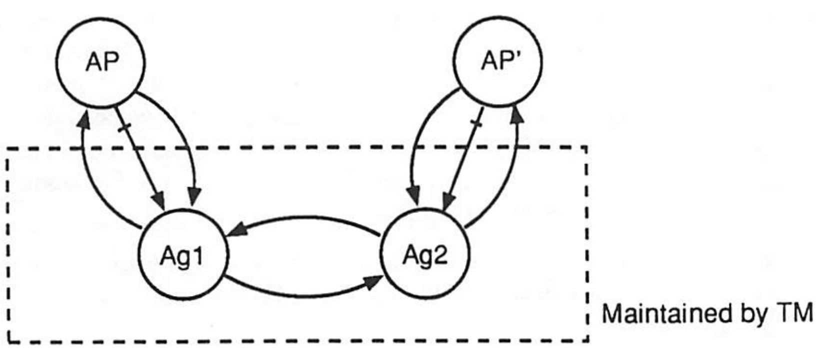


Figure 4 Agents and Dependencies on AP and ETM

Note that there are certain predefined dependency between the AP's and the TM. This relationship is defined as we conventionally understand how an AP and TM should work; if the AP fails, the agent maintained by the TM should also fail and vice versa. Also the TM agent can only finish after the AP agent has finished. Now when AP has done all its work, it issues the following call:

```
End_Transaction(Agent_id1);                            /* AP completed its work */
```

Then a finish event will be sent to the TM. The finish dependency thus gets resolved and Agent 1 may move further along on the event sequence as described in section 3. Since there is a mutual dependency between Agent 1 and Agent 2, if Agent 1 has finished first, it may trigger an event to Agent 2 to request a promise of committing the transaction instance. Since Agent 2 may not have finished, the TM therefore synchronizes the agents by holding the request event for Agent 2. Until AP' has proclaimed that it has finished (through the End_Transaction call), Agent 2 will not see the request-for-promise (req_prepare) event. The TM holds the events by setting up a list of preconditions for each agent at the time when dependencies are declared. Upon receiving an event for an agent, the TM checks the precondition list to decide if an agent can move on with that event. Notice that an event may satisfy and resolve some of the conditions, but the agent may still be required to stay since not all conditions are satisfied. Also, upon intercepting an event for an agent, the TM may further trigger other events to other agents because of dependencies. In the preceeding example, when the finish event is received by the TM for Agent 1, it triggers the request-for-promise event to Agent 2.

With this picture in mind, it is easy to see that a number of agents can be distributed onto multiple nodes. The only requirement is the ability to send an event across a node. This can be done by associating the receiver's address to the event we sent:

Message format = Sender's address : Receiver's address : Event

In our case, the address is composed of an agent identifier along with a system-wide node address.

6 Application Example

In the following example, we assume an application is distributed on multiple nodes and the transaction components are communicated in LU6 protocol which employs a flat transaction model. Each of these subtransactions accesses some database front end with a nested transaction model. The front end in turn talks to the back-end database server also in a nested transaction model. The back ends access the file

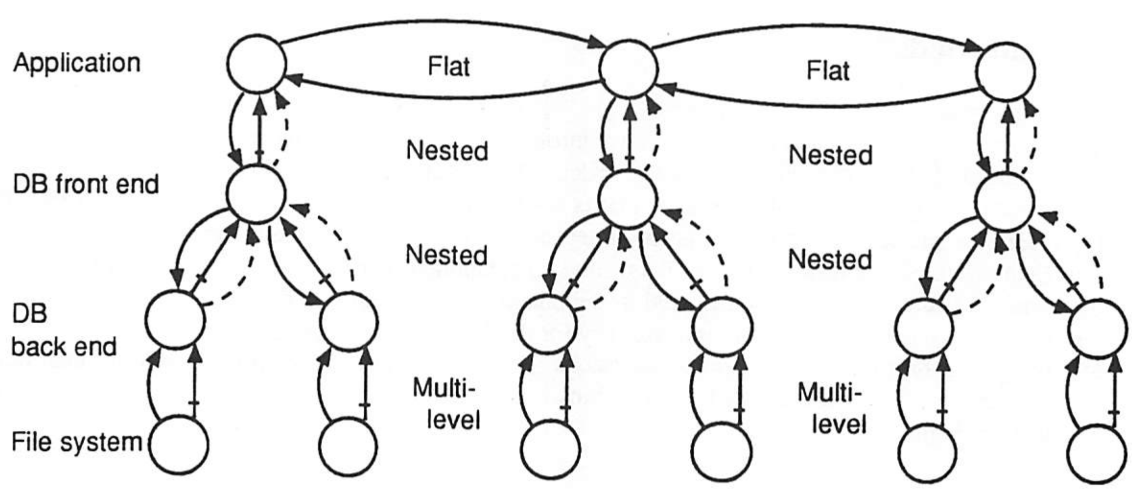


Figure 5 Mixed Transaction Models in a Distributed Database Application

systems by using a multi-level transaction model. A prototype of the ETM has been developed to manage the transaction tree shown in Figure 5. If one of the application transaction components aborts, the whole tree will abort except for the file system subtransactions whose committed effects, if any, can be undone

by a compensation. Note that the failure of either the database front end or back end will not cause the application to roll back. Therefore the database system has the capability to fail over to another process or another node if the database system failed in the middle of a data access.

7 Conclusion

The methodology proposed here provides a formal model of relating subtransactions together, thus allowing a flexible and organized way of constructing complex transactional applications. The event-based approach in the transaction manager gives a clear separation between the application's internal states and the transaction-related events. As a result, this approach avoids the cumbersome intermingling of application logic and transaction management rationality. The abstraction of agents, events and dependencies has brought us beyond the boundary of traditional transaction concept and moved us towards building reliable applications.

In this paper, the rollback recovery and commit-abort protocol can be handled even when multiple resource managers are involved. In order to focus on the discussion of the transaction manager, however, we have assumed that the concurrency control aspect will be handled by the lock manager of each individual resource manager. When multiple resource managers are involved, it is desirable to have a common service to resolve conflicts that are detected by the different lock managers. The concepts of conflict detection and resolution are being developed to tackle this issue and are the subject of continuous work. As mentioned in the preceeding application example, the idea of using dependencies to relate processes or systems together can be used to support hot-standby and other fault-tolerant related issues. More work such as journaling, archiving, and surveillance must be done in order to come to a complete solution. Nevertheless, we believe that we have a much more formal model to reason about fault-tolerant issues than today's ad-hoc approaches.

A prototype version of the event-based transaction manager has been built and is running and interoperating on both the Ultrix and VMS platforms today.

8 Acknowledgement

Dieter Gawlick has stimulated us in many ways throughout the course of this project. The idea of primitive dependencies was originated by Johannes Klein. He and Edward Lu are responsible for the creation of a generic event synchronization module that is used in the transaction manager. Ron Obermarck and Dora Lee have pushed forward the concept of separating conflict detection and conflict resolution. Lee and Klein together have produced a conflict resolution component which can be used along with the ETM prototype. Edward Chang has contributed in the communication layer of the transaction manager; as a result, we have accomplished locale-transparency for the TM-TM communication. Thanks to Bob Taylor for giving us technical guidance in many occasions. Thanks to him and Alberto Lutgardo for their management support. Special thanks to Janet Collins for her patience and support; without her this paper would not be legible.

Referrence

[1] Gray, J., *The Transaction Concept: Virtues and Limitations*, Proceedings of IEEE, 1981

[2] Moss, E., *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, 1985

[3] Garcia-Molina, H., Salem, K., *Sagas,* Proceedings of ACM SIGMOD, May 1987

[4] Weikum, G., *A Theoretical Foundation of Multi-Level Concurrency Control,* Proceedings on PODS, 1986

[5] Bernstein, P., Hadzilacos, V., Goodman, N., *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, 1987

[6] Rusinkiewicz, M., Elmagarmid, A., Leu, Y., Litwin, W., *Extending the Transaction Model to Capture more Meaning,* Proceedings of ACM SIGMOD, March 1990

[7] Johnson, J., Laing, W., Landau, R., *Transaction Management Support in the VMS Operating System Kernel,* Digital Technical Journal, 1991

[8] Klein, J., *Advanced Rule Driven Transaction Management,* Proceedings of COMPCON, 1991

[9] Klein, J., *Coordinating Reliable Agents,* submitted for publication

[10] Mohan, C., Lindsay, B., Obermarck, R., *Transaction Management in the R* Distributed Database Management System,* ACM Transactions on Database Systems, December 1986

[11] Andrade, J., Carges, M., Kovach, K., *Building a Transaction Processing System on UNIX Systems,* Proceedings of the UniForum Sixth Annual International Conference, February 1989

[12] X/Open, *Distributed Transaction Processing: The XA Specification,* April 1990

[13] Cheng, E., Obermarck, R., *The DEC Activity Recording Component (DECarc),* Technical Report, Digital Equipment Corporation, September 1990

The project team (**Edward C. Cheng, Edward Chang, Johannes Klein, Dora Lee, Edward Lu, Alberto Lugardo, Ron Obermarck**) belongs to the Activity Management Group (AMG) of Digital Equipment Corporation. AMG is responsible for the products and services which manage data and activities in a complex computing environment. The project team is made up of engineers and researchers from the areas of transaction processing, database management, distributed system processing, journaling, and authorization subsystem. Numerous publications in these areas have been offered by individuals of the team. The focus of this team is on the event synchronization (ES) technology of distributed computing. With the ES component, users can separate the application logic of their programs from the complicated sychronization protocol (such as 2PC). The building of distributed applications therefore becomes more straightforward. The interoperability of different protocols from different vendors are also made possible.

The event-based transaction manager presented here is a result of applying the ES technology in the transaction management environment.

Emerging Hypermedia Standards

Hypermedia Marketplace Prepares for HyTime and MHEG

Brian D. Markey
Digital Equipment Corporation
Multimedia Engineering
markey@wrksys.dec.com

Abstract

The *International Organization for Standardization* (ISO) and the *American National Standards Institute* (ANSI) are currently sponsoring two major projects relating to hypermedia.

ISO/IEC JTC1 SC2/WG12[1], known collectively as the Multimedia and Hypermedia Information Coding Experts Group (abbrev. MHEG), is developing a standard titled *Coded Representation of Multimedia and Hypermedia Information.*[2]

ANSI group X3V1.8M[3], known collectively as the Music Information Processing Standards (MIPS) committee, is developing a hypermedia document interchange standard titled *HyTime/SMDL.* HyTime has been officially accepted as an ISO project as well, following a successful new project ballot by ISO/IEC JTC1.

This paper describes the history, technical orientation and status of the MHEG and HyTime projects, as well as their relationship to multimedia (e.g. MPEG[4]) and document interchange (e.g. ODA[5] and SGML[6]) standards. The relationship between the standards is also explored, with emphasis on appropriate applications and situations where they can be used together in a complementary fashion.

The HyTime and MHEG standards are currently under development and will not be completed for some time. The final, published standards may vary in some important respects from the details given in this paper.

1. The Evolution of Hypermedia Standards and Applications

Multimedia is a term which has as many definitions as there are people who use it. ECMA defines it as *"a structure mixing various types of content like text, graphics, digitized speech, audio recording, picture animation, film clips, etc"*. Hypermedia is characterized by the ability to access multimedia information with explicit links. [7]

---

[1] Joint Technical Committee (JTC) 1 is the ISO/IEC body which is responsible for information technology standards.

[2] The name of this standard may change, so it is generally referred to as "the MHEG standard".

[3] Text Processing: Office and Publishing Systems, X3V1 is a technical committee under ANSI Accredited Standards Committee X3, Information Processing Systems.

[4] Proposed standard ISO 11172: Coded Representation of Picture and Audio Information as drafted by ISO/IEC JTC1/SC2/WG11: Motion Picture Experts Group (MPEG).

[5] ISO 8613: Office Document Architecture (ODA).

[6] ISO 8879-1986: Information processing - Text and office systems - Standard Generalized Markup Language (SGML).

[7] Definition from MHEG working document "S.3".

The notion of explicit links comes from hypertext applications. This association leads to a more obvious definition of hypermedia: hypertext with the addition of multimedia data types. Unfortunately, this view places conceptual limits on hypermedia which would relegate it to a few unexciting applications.

As work progressed on standardizing certain aspects of hypermedia, it became apparent that the concepts which hypermedia embodies have application that almost exceed the boundaries of imagination.

1.1 Scope of Hypermedia Standards

The marketplace tends to react to standards in one of two ways: standards either act as a catalyst for new products and technologies or they are ignored. Obviously, the developers of standards would prefer that their work was a positive influence. With this in mind, most standards bodies eventually wrestle with the question "*what is appropriate to standardize and what is not?*". In the software arena, where progressive thought is the norm, standardizing applications semantics or processing models is shunned. Bold and often unanticipated methods arise in a marketplace that's free from the constraints of standardized mediocrity.

While standards bodies typically do not define applications, they do define certain elements of the framework, such as data interchange formats, that applications will use. This facilitates the software vendor's desire to create innovative products, while allowing users the flexibility of open environments and shared information.

1.2 Hypermedia Standards Lead Product Development

Information technology products traditionally precede standards. Often, standards appear in final form years after the products come to market. The technology used by a particular software product may become the basis for a *de facto* standard or the technologies of several existing products may be combined. If more than one technology is used in a standard, and if there is a degree of conflict between technologies, then a standard is likely to emerge that is not wholly compatible with any of the base technologies. Software users suffer as a result, being unable to exchange information between applications because of prejudicial implementations of a standard by different vendors.

In the hypermedia software market, no clear product (or de facto) standards have emerged. In fact, there are relatively few hypermedia products available, and while some software vendors can claim dominance on a particular platform, none have the level of acceptance some word processor or spreadsheet applications enjoy.

Much of this is due to a marketplace which is not totally convinced it needs hypermedia applications to solve its problems. The only thing most people seem to agree on is that hypermedia is useful in the area of *Computer Aided Instruction* (CAI). Course materials in CAI applications are prepared using software known as *Authoring Systems.*

The students, for whom the output of the authoring system is intended, may use a different piece of software, and possibly even a different piece of hardware, to *render* (or present) the course materials. Often, more rendering than authoring systems are required (more students than teachers) and thus there is pressure to reduce the cost of the rendering equipment. When there is a significant gap in price, there is usually a corresponding gap in technology. In fact, it is likely that the authoring and rendering systems would be on completely disparate platforms (operating system, CPU, video technologies, etc.). Thus the need for standards governing data interchange formats.

1.3 New Applications of Hypermedia Systems

Hypermedia has broader applications than authoring/rendering, and these applications are just beginning to be understood. The prevailing view among computer scientists is that object-oriented hypermedia provides for the integration and manipulation of a wide variety of data types in an associative manner. Virtually any process can be represented using hypermedia techniques, and therefore the actual number of applications is almost limitless. It is acknowledged that authoring systems will continue to drive the hypermedia software market, at least for the foreseeable future. Due to the inevitability of other applications, however, it is deemed necessary to make interchange formats open and extensible.

The following sections cover two potential applications that might have been overlooked given a more limited definition of hypermedia.

1.3.1 The Insurance Adjuster Application

An insurance adjuster thumbs through a pile of papers on her desk, looking for the text of the police and accident reports filed several weeks ago following a two car collision. She finds the papers, and then turns her attention to a VCR and television monitor positioned on the other side of her office. She watches a brief videotape taken by police officers at the accident scene. Later she replays a cassette tape containing a recorded telephone conversation with the driver of one of the vehicles. As she reviews the materials associated with the accident, and thinks about the time she spent organizing them, she concludes that there must be a better way.

A vision forms in her mind of a computer workstation. On the screen of the workstation is an image of the two accident report documents. In another window, the contents of the video tape are displayed. Buttons appearing on a pull-down widget allow her to control the video image; she can stop, rewind and slow-scan the frames she deems important. In another window, a word processor allows her to enter information in a standard claim form. A digitized audio recording plays when she clicks on the appropriate icon. In her vision, all information pertaining to the claim has been organized into a single document (see section 2.1 for further discussions on hypermedia documents). She notes that, ideally, the document would be archived and retrieved as a single entity.

1.3.2 The Hyper-News at 11:00 Application

Jon is working late on a computer program in his office. In front of him, several windows are open on his workstation; some contain small pieces of C language source code, others output from his recalcitrant executable. The time display in one window is updated to read 23:00, and an alarm bell rings. Jon knows that this means that it is time to receive the news of the day.

After clicking on an Icon, Jon is given a menu of news stories. A piece about strife in his native Lithuania catches his eye, and he uses the mouse to select it. The face of an incongruously ebullient newscaster appears in one window, in another, a scene of police clashing with demonstrators. The workstation's built-in speakers vibrate to the chirpy tones of the newscaster's voice. At the

bottom of the display, the otherwise solemn text of the story scrolls by. Buttons allow Jon to display the text of related newspaper and magazine articles.

Jon decides he's better off finishing his program and, with an unceremonious click of his mouse button, relegates the news application to an icon.

2. Hypermedia Concepts

All hypermedia systems share certain characteristics: data formatting, synchronization, hyperlinking, etc. Before discussing the specifics of HyTime or MHEG, it is important to explore these characteristics in some depth.

2.1 Hypermedia Documents

In document processing parlance, a *document* is defined as *"a collection of information that is processed as a unit"*. As illustrated in the insurance application, the collection of information does not have to be confined to character text (as generated by a *text editor* or *word processor*). The term document is applied to hypermedia in roughly the same form as it is in document processing standards such as ODA or SGML: the document is a *representation* of information intended for processing by some application which prepares it for *presentation* (i.e. human perception).

In document standards, documents contain *text*, which includes *data* and *markup*.[8] Data, as will be shown in the next section, covers just about every imaginable form of coded information. Markup is defined as text that is added to the data of a document in order to convey information about it. One use of markup is specifying the structure of a document: chapters, paragraphs, headings, etc. Markup is also useful when the author wants to specify some intent in the representation of the document which should have a corresponding effect on the presentation. For instance, the author can express his intent that a section of text should be presented using bold-face type. Defining standardized markup is one of the most important elements of developing an interchange standard.

A hypermedia application may be used for the authoring of hypermedia documents, for rendering them, or a combination of both. Further, the application may simply accept the document as input for other processing. This would apply to situations where the document is intended to *model* data rather than directly represent it. Project scheduling is one application that might use a document in this way.

2.2 Embedded Multimedia Data

To many, the term *text* implies *character data*, but in document processing standards it has a broader definition which would include any data which might be encoded into the document. Besides character data, graphics, digitized video, audio, MIDI[9] and other data types associated with multimedia applications may be present in the document. Certain documents might not include any character data.

Often, the format of data that is embedded in a document is dictated by other applications which read or write that particular format. The requirements of the hypermedia application determine whether the exact format of the data or only the data type (e.g. MIDI) must be specified. Within a document, multimedia (and other unconventional data types) are declared using a mechanism known as *content notations* which

[8] The terms *markup, text* and *data* are used as defined by the SGML standard (discussed in section 3.2). Other standards (such as ODA) use different terms and a somewhat different model, but a discussion of these matters is outside the scope of this paper.

[9] Musical Instrument Digital Interface (MIDI) standard version 1.0 by the International MIDI Association (IMA).

provide information about the data format. The syntax required to define a content notation is specific to the markup language in use. Normally, the process involves the creation of a header which declares the data type and the data stream's length. In some instances, the header will contain additional information about the data format, which may allow the application to link to specific portions of the data stream.

Data in a hypermedia document can be organized into *objects* which are addressed as singular units of some content notation. For instance, a sequence of data may be defined as containing 360 frames of MPEG compressed video, and the sequence would be accessed as a single object by the hypermedia application.

The term *embedded* should not be construed to mean that the data must be part of the normal document stream. In fact, it is possible that data could be contained on any number of external devices including video tape, audio tape, slides, etc. Generally, only a description of the object must be embedded in the document stream, or reference must be made through a link (discussed in section 2.3) to a structure which describes the object.

2.3 Hyperlinks

Anyone who has ever written a report or paper that references other materials has used a primitive form of a link. This type of linking is known as the *bibliographic model*; a document makes reference to another document to supply additional information that could not be readily incorporated. In traditional publishing, the work of activating the link is left up to the reader of the document: the reader is told where to look for more information, and is expected to locate the book and section that are referenced.

Computer users would also be familiar with the next step in the evolution of links: on-line help. Most applications are now built so that when a user gets stuck he can activate a facility which will give him additional information about what he needs to do. Further, many applications use the notion of *contextual linking,* wherein the help information retrieved is relevant to the process being performed at the time help is requested.

In bibliographic references, the author making the reference has some notion of the scope of the *object* to which he is referring. For instance, when an author's reference specifies a range of pages in another work, they have imposed a bounding box on the referenced material. The view of the objects in the referenced work are likely to be quite different than the view the original author had in mind.

In computer programming, when control is passed from one location to another, generally there is a label at the target location. In this instance, the writer of the targeted piece of code knew it would be externally accessed and generated some unique identifier that could be readily translated into an address. If the bibliographic reference paradigm is extended to a computer application, it is clear that referenced data cannot be limited to that which has an identifier to establish its address.

Generally, the author of a document gives sufficient clues to resolve a link target address; he might refer to book x, chapter two, third paragraph. A problem arises, however, when the author of the referenced document changes it. If he happened to add a new chapter between chapters one and two, the reference to the old chapter two becomes meaningless. For any hyperlink model to be complete, it must address this type of problem.

The basic functional areas covered by any hyperlinking method may be summarized as follows:

- Demarcation of objects: the ability to organize data into elements which may be accessed as a single entity.

- Identification of objects: the ability to define a unique identifier for each object, such that it's location can be established.

- Linking: creation of links which reference identified locations.

2.4 Synchronization

In a hypermedia system, objects may be placed in hypothetical structures known as *events.*[10] Events imply that some action is to be associated with the object at some particular point in time. The action taken is determined by an application provided *method.* In some systems, the method is implied by the object type. In others, the name of the associated method is embedded in the object representation.

Events are typically synchronized in one of two ways: relative to each other or relative to a schedule. For example, you could say that event B occurs after event A, or you could say that relative to schedule *t,* event A takes up time quanta 3 through 5, and event B quanta 6 through 8. Both methods of synchronization must be provided for the model to be complete.

The synchronization information represented in a document and the resulting presentation are not necessarily identical. Due to physical limitations such as system performance, the presentation of synchronized objects may require certain compromises.

There are several levels of synchronization in typical multimedia/hypermedia applications: some apply to object level structures, others to physical data streams (e.g. audio, video). Physical data within an object may use its own synchronization method, either implied or explicit. For instance, audio and video that are embedded in the same stream may be synchronized using a proposed MPEG encoding. The method associated with the object would need to be able to perform the necessary synchronization algorithm.

Ideally, there would be some mechanism for the method to communicate synchronization back to the application. For instance, a separate object might hold foreign language subtitles which would need to be synchronized with the MPEG audio/visual object.

3. HyTime

As defined in the MIPS committee documents, *"HyTime is a standardized infrastructure for the representation of integrated open hypermedia documents".* The concept of integrated open hypermedia embodies three basic elements:

- Integrated: all information is linkable, whether or not it was explicitly prepared for linking

- Open: link addressing is independent of the file management or network architectures of particular platforms

- Hypermedia: combines both hypertext (hyperlinking) and multimedia capabilities

3.1 The History of HyTime

A group was convened in 1986 under the auspices of ANSI committee X3V1 to develop a standard for the interchange of music processing information. The group was known as the Music Information Processing Standards (MIPS) committee, and the standard it developed was titled *"Standard Music*

[10] Usually, the synchronization information associated with an event is an attribute of the object, but for purposes of this discussion it is better to think of the event as enveloping the object.

Description Language" (SMDL). The need for SMDL first rose in the music publishing industry. They needed a markup language for preparing musical scores for publishing. Music publishing poses even more problems than publishing mathematical texts: the symbol set is infinitely variable and is almost impossible to represent as a typical font, plus the number of symbols is very large.

Music description languages generally came in two flavors. Some are fine for representing music for *computer assisted performance.*[11] Others are suitable for music publishing. The performance oriented languages usually are not suitable for printing scores. The music publishing languages are often incompatible with other document processing standards and they usually have limited (or no) capability for performance. The MIPS committee decided that they would address the needs of publishers and performers alike in SMDL.

One of the early revelations to strike the MIPS participants was that music has a very robust and stable timing model (this notion is discussed further in section 3.3). Further, music has several constructs which fit the hyperlinking paradigm quite well (e.g. repeats, codas). Since synchronization and hyperlinking are two essential elements of hypermedia, the work of the MIPS committee progressed naturally toward HyTime.

Now, SMDL exists as an application of HyTime, which in turn exists as an application of SGML (which is discussed in the next section).

3.2 The Coding of HyTime Documents

Charles Goldfarb of the IBM Almaden Research Center was the editor for the document processing standard known as *"Standard Generalized Markup Language"* (SGML). Dr. Goldfarb was asked to chair the MIPS committee as well, so SGML became a natural selection for the markup language upon which SMDL (and later HyTime) would be based. Further, SGML support was growing among publishers, and many SGML applications were starting to appear in the marketplace.

SGML is extensible, in that application specific markup can be defined using a mechanism known as *Document Type Definitions* (DTD). In a typical SGML application, a document is combined with one or more DTDs which specify how the document is to be processed (the combination is known as a *document instance*). The document instance is processed by an SGML parser which lexically analyzes the document to distinguish markup from data. The SGML parser notifies an SGML application of any markup that it locates. The application is responsible for taking the appropriate action (the action is not defined by the standard, but the mechanism for binding the action to a particular syntax is).

Although the demand for HyTime is still driven in large part by SGML users, there is the realization that HyTime embodies an architecture (or framework) from which other hypermedia standards, based on different document processing technologies, could emerge. For instance, the idea of ODA encodings for HyTime constructs is being actively explored.

3.3 HyTime Synchronization and Timing Model

> Stage hands finish last minute preparations, positioning scenery as actors wait nervously for their cues. Lighting technicians dim the house lights and bathe the stage with subdued tones from border spots. In a pit below the stage, 30 musicians watch as the conductor lifts his baton. The production of Gilbert and Sullivan's *H.M.S Pinafore* begins...

---

[11] Even MIDI could be used for this purpose, although it would be analogous to writing computer programs in hex.

Using a variety of technologies available today, it's possible to describe the production using a HyTime document. An application would use the document as the database for a variety of control functions. The lighting might be controlled using MIDI. A sampled foley database could provide sound effects. The actors may receive queues from Teleprompters, displaying text objects synchronized to the music. The music might even come from a rack of samplers and a MIDI sequencer.

As discussed in section 2.4, objects may be placed in bounding structures known as *events* which occur at some point in an ordered list known as a *schedule*. Synchronization is the alignment of events with the schedule, or along an axis which represents time. While HyTime foresees the legitimacy of applications which have no time dimension, it offers a number of constructs for the majority of applications that will have one.

Music has always used the concept of virtual time. For instance, if a musician refers to a "half note", no duration in real time is expressed or inferred. The term "half note" is in units of virtual time. Only the note's relationship to other notes - whole notes, quarter notes, eighth notes - is certain. Indeed, a half note is half the duration of a whole note, but if asked to attach a real time duration to that particular event the musician would point out that there is insufficient information. The actual (or real) duration of the note is determined by other factors such as the time signature and the tempo. The tempo may, in turn, be a relative figure of merit (virtual). It is common for composers to use terms like *slow* and *fast* to describe the tempo, leaving the interpretation up to the conductor. Assuming the conductor is a person and not a cesium clock, there will be other variations as well.

The concept of virtual time has application beyond music. For instance, if the milestones in some project schedule are represented as objects in a HyTime document, and their relative magnitudes are understood (e.g.: "it takes twice as long to debug program x than it does to write it"), then the schedule dates are easily recalculated given the real time duration of one of the objects.

Music also has the concept of indefinite duration. While the actors in our mythical operetta complete their bows, the orchestra continuously repeats some sixteen bar pattern. When an external event occurs, in this case the stamina of the adoring audience is exhausted, the actors leave the stage, which is the triggering event indicating the musicians should move to the end of the score. In HyTime, the triggering event might be the completion of some other event (the second video clip is shown after the first one completes), or it might be the result of some external stimulus such as a mouse button click.

Musicians, or at least musicians who prefer to keep their jobs, use the conductor as their source of timing information. The conductor wields the baton, and all attention had better be focused upon it. HyTime borrows the concept of the baton to indicate the function which maps virtual time to real time. If more than one baton exists, one is designated as the master unto which all other batons are subordinate.

As shown in sections 3.5 and 3.7, the baton is not limited to the scaling (or conversion) of temporal coordinates.

3.4 HyTime Hyperlinking

HyTime hyperlinking is based upon link capabilities that are required for all document processing and are thus already part of the SGML standard. The functional areas of the hyperlinking model, as discussed in section 2.3, are fully covered by HyTime and SGML.

Each HyTime hyperlink has at least two *link ends*. One link end is the element in which the link reference occurs, the other end is the identified location (known as an *anchor*). In some instances, there

may not be an identifier in place to which reference can be made (this situation is discussed in section 2.3). To solve this problem, HyTime provides *location pointers* which assign ID attributes to arbitrary locations. A location pointer serves as a pointer to an anchor. Link references can be made to the location pointer instead of directly referencing the anchor. In other words, location pointers provide a level of indirection similar to the pointer constructs in C. This is useful if the referenced content has been moved or otherwise reorganized. It is also useful in cases where the object being referenced was not demarcated as such in the original document.

HyTime uses the bibliographic reference model (discussed in section 2.3) for hyperlinks. There are several types of hyperlinks, each with a specific purpose and scope:

- context links: A link with two link ends, both of which must be addressable within the same *context*. This link type is useful for intradocument links such as footnote references.

- independent links: A link with two or more link ends, and whose addressing is not confined to the current context. Uses include references to multiple documents.

- property links: A variant of the independent link, it is useful for referencing content in which anchors cannot be established (such as when there is no write access to the document being referenced). Applications include reviewer's annotations to a document.

3.5 HyTime Cosm

In classical mechanics, the position of any object is expressed using six axes; three for position and three for momentum. In HyTime, objects exist in an application defined n-dimensional space known as a *cosm.*[12] The traditional three dimensional spatial model, or even a four dimensional model (spatial and temporal) was avoided so that HyTime could be generalized for any application.

The cosm is a hypothetical bounding structure. In order to be useful, it must be represented using some coordinate system known as a *phase space*. The axes of the phase space are not required to be orthogonal. Consistent with the synchronization model, the coordinate system for any axis may be real or virtual, i.e. it may be based either on known units of measure (inches, seconds), or on relative sizes and durations.

Multiple phase spaces may be defined for a cosm. The phase spaces need not have the same number of dimensions, nor do they need to share coordinate systems. *Projection* is the process of converting events from one phase space to another, such as virtual coordinates to real. Projection is used in circumstances where the representation phase space is different from the presentation phase space. The rules for projection are determined by *scaling directives,* which occur in *baton* elements.

HyTime units of measure are specified in a hierarchy, whereby one may be defined in terms of another. The basic units of measurement are termed *Standard Measurement Units* (SMU). Some commonly used SMUs exist as predefined public identifiers in the HyTime DTD:

SIsecond	System Internationale second
NISTinch	US National Institute of Standards inch
virtime	virtual time unit
virspace	virtual space unit

[12] From cosmos: an orderly harmonious systematic universe [Webster's New Collegiate Dictionary]

Multiples or submultiples of SMUs may be defined as *HyTime Measurement Units* (HMU). HMUs may also be defined as granules of other HMUs. There is no limit on the levels of conversion.

The need for this feature is obvious. A HyTime application which does project scheduling is more likely to express units in terms of days or weeks than in terms of SIseconds. A molecular modeling program would have little use for the NISTinch; granules for angstroms or nanometers would be more appropriate.

3.6 HyTime Alembics

Alchemists, and later chemists of an even more entrepreneurial disposition, used an apparatus called an alembic in the distillation process. The alembic is something that refines, filters or transmutes its inputs.

This is exactly the nature of the alembic in HyTime. Objects are "passed through" an alembic and are thus transmuted in the process. Among the many uses of alembics are:

- change color maps (apply color filters through changes in palette)
- display a range of video frames at a slower than normal speed
- modulate digital audio objects (change amplitude, add reverb or vibrato)
- scale a particular phase space

Alembics may be used in combinations known in musical circles as *patches*. To continue the analogy a step further, consider the following example in terms of a HyTime document:

> In the early days of electronic instruments, the musical synthesizer existed as a group of modules, each with a specific function (oscillator, voltage controlled amplifier, voltage controlled filter), that was controlled by a piano-like keyboard. In order to change the sounds the instrument produced, controls were manipulated on each module. Further, the order of processing by the modules could be modified by placing short lengths of wire, known as *patch cords,* between modules.
>
> Imagine a HyTime application that implements, using digital audio technology, the corresponding analog functions of the individual modules. A piece of music is encoded in SMDL (remember that SMDL is an application of HyTime), and this is used as the input to our synthesizer application.
>
> Along with the music coded in the document, objects are defined which will produce specific waveforms. Further, alembics are defined which will modify the waveforms, by changing the frequency content and amplitude in "real time".

3.7 HyTime Scaling

Batons, as stated in section 3.5, are elements which contain scaling directives which define rules for the projection of one phase space to another. As with alembics, the applications of the baton are virtually unlimited:

- change tempos in music, as in accelerando or ritardando

- control the pause between slides in a presentation, allowing for different lengths of narration
- scale a sequence of graphic images so that they all use the same amount of display space

The scaling directives allow the author, in the representation of the document, to be as specific as he wants to be about the presentation. In some instances, the specification of the scaling will be such that there is really only one way to correctly render the document. In others, any number of renditions could result. The author's intent is preserved throughout all possible renditions.

To illustrate this point, let's return to the example of the music encoded as a HyTime/SMDL document. The composer of the music may specify that the music is to be played fast, which allows for a large number of interpretations, or he may specify that it is to be played at a precise tempo of 120 beats per minute.

4. MHEG

Jennifer chooses whales as the topic of her biology class paper. At home, she selects a CD-ROM labeled *Encyclopaedia Generica volume W-Z* from a series of plastic boxes. She presses the button on the CD/CD-V/CD-I component in her entertainment system, and it slides open a drawer into which she inserts the disk. A menu of topics is displayed on her television monitor and with a twist of the jog-shuttle dial on her remote control, she selects "Whales". Text appears on the screen and beside it a graphic image depicting the whale's anatomy. A row of buttons is displayed below the text which allow her to select from related topics ("Aquatic Mammals", "Moby Dick", "Plankton", "Star Trek"). Another button plays a video segment of divers following whales in their natural habitat, complete with recorded whale sounds.

The CD-ROM encyclopedia is today's reality, but the home interactive hypermedia appliance is still in the future. When it does appear, it's possible the CD-ROM will be coded in the data format defined in *Coded Representation of Multimedia and Hypermedia Information*. With such a formidable moniker, the standard is more commonly referred to as MHEG (as it shall be for the remainder of this document), named after the ISO working group which is developing it.

MHEG is ideally suited to applications such as the hypermedia encyclopedia. MHEG objects are encoded to be presented in real time. No complex parsing is required, and all links are fully resolved: the exact location of an object being linked to is specified in the object associated with the link.

MHEG expresses objects in final *nonrevisable* form and is therefore unsuitable for highly interactive authoring applications. Due to its inherent real time nature, however, MHEG is quite suitable for representing the output from authoring systems.

4.1 The Coding of MHEG Objects in Documents

MHEG is a standard for representing objects, not a document processing and interchange standard. The representation of MHEG objects is intended to be independent of all document encoding. MHEG provides rules regarding the structure of objects which should accommodate their encoding in any notation.

In the completed standard, MHEG will provide a base encoding using ASN.1.[13] Since ASN.1 is presently used in other standards, including ODA, there is a natural progression for extending applications to incorporate hypermedia capabilities.

The completed standard will also provide a DTD for representing MHEG objects in SGML documents. With efforts at *harmonization*[14] underway, it is likely that MHEG object structures will also be uniformly supported by HyTime.

4.2 MHEG Objects

MHEG allows many data types, including those unique to multimedia applications, to be represented as objects. Each object is manipulated as a single entity in a manner consistent with the definition in section 2.1.

Basic MHEG objects contain information of one particular type: text, graphics, video, digital audio, etc. Basic objects may be combined to form *composite objects*. Composite objects may also contain other composite objects. Within a composite object, each of the component objects have a predefined relationship: they are either synchronized in some manner or they are related in terms of hyperlink navigation.

4.2.1 MHEG Object Types

There are four MHEG object types: *Input, Output, Interactive* and *Hyperobject:*

- <u>input object</u>: composite or basic objects which represent input from the user (ex: buttons activated, menu selections, input to forms or input to edit fields)

- <u>output object</u>: basic or composite object that is intended to be presented in some way to the user (ex:text, graphics, images, audio or audiovisual sequence)

- <u>interactive object</u>: composite of input and output type objects (ex: menu plus user's menu selections, or form plus user's inputs to form)

- <u>hyperobject</u>: composite object consisting of input and output objects plus explicit links between them.

Interactive objects and hyperobjects are always composite objects, in that they exist only in conjunction with other objects.

4.2.2 MHEG Object Classes

MHEG objects are categorized by classes, where the objects in a class share behavior, characteristics and functions which manipulate them. Each object belongs to one class. A class is associated with one *representation medium* which defines the nature of the information in coded form (audio, video, text, etc.). A given object is an *instance* of a particular class.

There are classes associated with basic objects (input, output) and with composite objects (interactive or combinations of multiple input and output objects).

---

[13] ISO 8824: Specification of Abstract Syntax Notation One (ASN.1) and ISO 8825: Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).

[14] A term used by standards developers (particularly in ISO) to connote cooperation between groups with the intent of preventing contradictory mechanisms from being standardized.

Each class has a list of *attributes* associated with it which govern the behavior of objects in that class. Refer to section 4.2.4 for more details regarding class and object attributes.

The MHEG standard defines specific object class names and attributes which correspond to most of the significant data types which are in use today. The object type examples given in 4.2.1 would each be associated with a particular class.

In some cases, data of one class may be encoded using a number of different algorithms. For instance, digital audio has a number of standardized encodings, and more are likely to emerge. In these cases, the *type* field in the object descriptor (see 4.2.4) indicates the algorithm that should be used.

4.2.3 MHEG Object Methods

Each object will have associated with it a *method,* or a procedure which controls the presentation of the object. The attributes associated with the object form the parameters of the method.

The method would normally be associated with a class of objects, such as digital audio. In this case, the method might be a routine which prepares the data for output to a DAC.

MHEG makes no attempt to define the nature of the method: it is a system and application design issue. Methods may be entire programs, system services or routines in an application. Some methods may require interrupt driven interfaces. In some circumstances, the method might need to be re-entrant (to handle multiple streams of the same object class simultaneously).

4.2.4 MHEG Object Representations

An MHEG object may be represented using a combination of *All-object* representation attributes and *Class* representation attributes. The All-object attributes include:

- Interchange identifier
- Object class name
- Object descriptor (name of object, version, date, etc)
- MHEG version
- Profile
- Object status

The object class attributes are specific to the class (and thus the data encoded in the object). They provide all the information the method requires to render the object. The details for the attributes have not been completed for all classes, but a couple of examples follow:

- For audio objects: includes coding method, sample rate, volume, balance, direction (forward, reverse) and recording duration
- For still picture objects: includes coding method, position (x,y), frame size and object virtual space limits

4.3 MHEG Synchronization and Timing Model

MHEG supports the following synchronization modes:

- atomic serial synchronization: Two component objects of a composite object occur one after the other without a delay between them.

- atomic parallel synchronization: Two component objects of a composite object are synchronized with regard to the same reference origin time.

- elementary synchronization: Offers two modes of operation, sequential or parallel.

  In the sequential case, two component objects (A and B) and two time values (T1 and T2) which are associated with the object are specified. Object A is activated at time T1, relative to the activation time of the composite object. Object B is activated at time T2 after the end of the event associated with object A.

  In the parallel case, two component objects are synchronized with reference to the activation time of the composite object. As with the sequential case, two objects (A and B) are specified, as well as two time values (T1 and T2) associated with the objects. Object A will be activated at time T1, relative to the activation time of the composite object. Object B will be activated at time T2, also relative to the activation time of the composite object.

- conditional synchronization: Allows the activation of the presentation of objects to be dependent upon the state of *mark state variables.* An expression (which can include a combination of mark state variables, relational operators and boolean operators) is evaluated, and if the value of the expression is true, then the presentation of the object is activated.

- cyclic synchronization: Allows events to be synchronized to some periodic event, such as a clock tick.

- chained synchronization: Allows basic objects to be chained together into a new composite object for presentation and synchronized with other streams. For instance, consider a composite object consisting of a video clip object and a chain of text subtitle objects. A *method* associated with the presentation of the video object is invoked. The method also updates some mark state variable as it proceeds, which causes the method associated with the presentation of the subtitles to transverse the chain.

Some objects may have synchronization information embedded within them that is outside the scope of MHEG synchronization (as discussed in section 2.4). Often, these objects are really composite objects which combine two data types (such as audio and video), but for the purposes of MHEG they are treated as basic objects. In such cases, the method associated with the object must provide the synchronization.

4.4 MHEG Hyperlinking

Objects may be defined which have the specific purpose of aiding in the navigation of hyperlinks. A *hyperobject* is a composite object where explicit links exist between an input object and some other object.

For instance, an application may specify that an audio segment is played when some push-button is activated. A hyperobject would be declared which has as its component objects the audio segment and the push-button. The attributes for the push-button would be defined such that the activation of the audio object is predicated upon the button's activation.

4.5 MHEG Coordinate Systems

MHEG uses the concept of virtual space in a similar manner to HyTime. The axes of MHEG correspond to the traditional spatial and temporal domains. The mapping of virtual domain coordinates to real ones is dependent upon the application.

The object attributes include coordinates for placing the object in a spatial or temporal domain, relative to other objects in that domain. As with synchronization, composite objects control the domain which applies to their component objects. Thus, there is an inheritance of domain from parent objects, and sibling objects share domain. Once an object inherits a domain, it may treat it as a virtual domain onto which the object may map any coordinate system or scaling.

At the top of the object hierarchy, *root objects* are placed in a generic virtual space. Their position in that space is determined by the attributes of the root objects. All root objects share the same generic space.

5. Summary

At first glance, HyTime and MHEG may appear to be conflicting standards since both define formats for the representation of of hypermedia information and are intended to facilitate the interchange of such information between applications. In fact, in some specific applications, MHEG and HyTime might assume complementary roles.

Each standard has a limited scope and purpose. In many instances, the scope of one supplements the other. For instance, HyTime may not be suitable for real time rendering of documents, whereas MHEG is by definition. MHEG, on the other hand, has minimal capability for document revision, making it a suboptimal choice for highly interactive authoring applications.

Figure 1 demonstrates the potential for a symbiotic relationship between the two standards using a common application structure.[15]

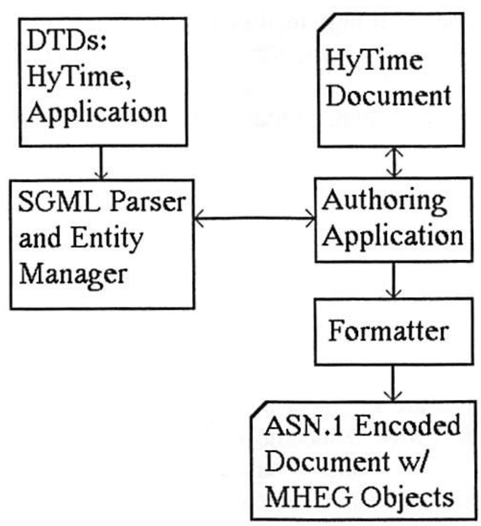


Figure 1: Application Structure Using HyTime and MHEG

[15] The application structure shown in Fig. 1 assumes a symmetry of function between HyTime and MHEG that presently doesn't exist. This matter requires further study by the working groups before it becomes reality.

Acknowledgements

I am indebted to my Digital colleagues, particularly Richard Hovey, Massimo Boano, Carol Young, Leszek Kotsch, Greg Wallace and Dick Bergersen for their comments and contributions. John Morse and Marshall Goldberg provided me with the opportunity to write this paper as well as ample encouragement to complete the task. Various presentations on the subject of HyTime by Charles Goldfarb, Steven Newcomb and Neill Kipp were invaluable, as were MHEG presentations by Hiroshi Yasuda and Francis Kretz.

Bibliography

ISO 8879-1986: "Information processing - Text and office systems - Standard Generalized Markup Language (SGML)"

ISO 8613: "Office Document Architecture (ODA)"

ISO 8824: "Specification of Abstract Syntax Notation One (ASN.1)"

ISO 8825: "Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)"

ISO 11172: "Coded Representation of Picture and Audio Information" [MPEG committee draft]

ISO 10744: "Hypermedia/Time-based Structuring Language (HyTime)" [MIPS committee draft]

ISO/IEC JTC1/SC2WG12 working documents S.3: "Coded Representation of Multimedia and Hypermedia Information" and E3: "Experimental Interchange of Multimedia and Hypermedia Information Objects"

International MIDI Association (IMA): "MIDI 1.0 Detailed Specification", Document Version 4.1.1, February 1990

Douglas C Engelbart, "Knowledge-Domain Interoperability and an Open Hyperdocument System", Bootstrap Project, Stanford University pp14 (Copyright © 1990, Association for Computing Machinery)

Donald R. Sloan ,"Timely Thinking: An Introduction to HyTime", Music Department, State University of New York at Binghamton 8pp. Personal contribution to MIPS committee, document no. X3V1.8M/91-12.

Robert L. Travis Jr., "CDA Overview", 8pp. Digital Technical Journal, Volume 2 Number 1, Winter 1990

Biography:

Brian Markey is a Principal Software Engineer in Digital's Multimedia Engineering Group. Mr Markey represents Digital at HyTime and MHEG meetings, as well as meetings of other standards committees which are doing related work. Mr. Markey has an extensive background in multimedia, personal computer software and hardware and was formerly the project leader for Digital's Pathworks for VMS product.

Prior to joining Digital, Mr. Markey did digital audio research and product development for his own company (CompSync Incorporated) and was an instructor in the State-of-the-Art Engineering program at Northeastern University. He has also worked as a session musician, recording engineer, producer and technical consultant to the music industry.

Multimedia Presentation System "Harmony" with Temporal and Active Media†

Kazutoshi Fujikawa, Shinji Shimojo, Toshio Matsuura,
Shojiro Nishio, and Hideo Miyahara
Osaka University, Japan
fujikawa@ics.osaka-u.ac.jp

Abstract

This paper proposes a multimedia presentation system *Harmony* which can deal with *temporal* and *active* media such as motion video, computer animation, and music sounds. *Harmony* is based on the notion of a *hyperobject* which integrates a *hypertext* system with an *object-oriented* framework. Each *object* is considered a *node* as is usual in hypertext systems, and the relations between objects are represented as *links* between nodes. *Harmony* extends the ordinary notion of a link. A link in *Harmony* consists of three components: *objects* connected by the link, *conditions* specifying when the link is navigated, and *messages* sent to the target object. In addition to such an extension, each object can include internal objects called *subobjects* which specify parts of an object and can become a source/destination of a link. Furthermore, the notion of a *group object* is introduced to represent a synchronization of parallel displayed media information. Through these extended notions of a hypertext model, temporal and active media can be easily handled through the description of temporal relations from the media in hypermedia documents. Based on the design of *Harmony*, we have implemented a prototype multimedia presentation system which deals with text, music, graphics, motion video, and computer animation as objects. *Harmony* consists of three subsystems: *link manager subsystem* (Harmony/LM), *user interface subsystem* (Harmony/UI), and *object-oriented database* (Harmony/DB), where a commercial object-oriented database was employed as the Harmony/DB. *Scenario viewer* was developed for displaying the structure of scenario in a tree graph which can reduce a cognitive overhead in a hypermedia document. The whole system was constructed based on C++ language.

1 Introduction

In addition to *static* and *passive* media such as text, image, and computer graphics, *temporal* and *active* media such as motion video, animation, and music sounds are now available on a workstation or a personal computer with the help of special hardware devices. By introducing these new media, we can create a richer multimedia presentation beyond the desktop presentation. *Hypertext*[1, 2, 3] is a very attractive and powerful candidate for authoring system of multimedia presentations. A hypertext consisting of multimedia information is called *hypermedia*. One can make a presentation with combined multiple media easily and effectively through linking different kinds of media information in a hypermedia system.

†This research was supported in part by the Scientific Research Grant-in-Aid from the Ministry of Education, Science and Culture of Japan.

For dealing with temporal media in the hypermedia system, the ordinary notion of a *node* and a *link* in the *hypertext* must be extended due to the following three observations:

1. The information carried by temporal media may change continuously, and because of this property, we need a mechanism that dynamically updates links attached to the media.

2. Contrary to passive media (i.e., in these media no action is expected unless something acts from the outside), temporal media are active and can act spontaneously. There are many sources of events in the active media such as motion video. For example, whenever a person in a motion video makes a spontaneous action, events corresponding to his motions are automatically generated. In this case, the user may try to capture these events as a source of a link and navigate the link to another node. To satisfy such requirements, the system should have the capability to handle active media.

3. *Parallelism* should be introduced in navigating and displaying multiple temporal media simultaneously. For example, there are many instances in which picture and sound are shown at the same time.

In addition to the need for handling temporal information, the conventional notions of node and link in the hypertext cannot handle complicated multimedia information, much less the growing number of multifarious media. There is a need for a more capable modeling methodology for hypermedia systems to cope with the increasing number of media. Furthermore, in developing such powerful multimedia systems, it is important and practical to introduce *transparency* when dealing with multifarious media. To achieve this transparency in a system implementation, there is a need to provide the same interface for multifarious media in order to realize a consistent system design methodology.

To provide a powerful abstraction mechanism for complicated multimedia information and to include parallelism in system modeling, the *object-oriented* framework[4] seems to be most promising. We have adopted the object-oriented framework in designing our hypermedia system to include temporal and active media, and to integrate differing media into a single system. Such hypermedia systems based on object-oriented paradigms shall be called *hyperobject* systems.

In our previous paper[5], we discussed the design methodology and its implementation of our developing hyperobject system *Harmony*. In *Harmony*, multimedia information including text, music, graphics and motion video are considered as objects and are integrated into a single system providing the same interface to different media. An object in *Harmony* is considered as a node in the usual hypertext systems. A relation between objects is represented as a link between nodes. Extending the ordinary notion of a link, *Harmony's* links consist of the following components: *objects* connected by the link, *conditions* specifying when the link is navigated, and *messages* sent through the link. Due to the extended notion of links, *Harmony* is capable of handling temporal and active media such as motion video and music by describing the relation between media in hyperobject documents.

However, the system presented in [5] is a rather primitive one and we have extended and improved *Harmony* in many aspects aiming at resolving the following key issues:

- implementation of a synchronization mechanism among displayed multifarious media information,

- implementation of a mechanism to handle partial information such as a person in a motion video (i.e., the function for handling *subobjects*).

We also introduced new features into the prototype system. Among these are

- reconstruction of the system based on the C++ language,
- refinement of the user interface system,
- introduction of a commercial object-oriented database VERSANT[1] as the database subsystem,
- addition of a new medium, i.e., computer animation,
- implementation of *scenario viewer* for displaying the structure of scenario.

The system architecture of *Harmony* is based on the *server/client model.* As the database server of *Harmony* we employed VERSANT, a commercial object-oriented database.

In this paper, we describe the design methodology of *Harmony* and its second prototype implementation. In the next section, we describe the features of *Harmony.* The hyperobject model is specified in section 3, and temporal and active media are discussed in section 4. In section 5, we describe the system architecture of *Harmony. Harmony* consists of three subsystems: *link manager subsystem* (Harmony/LM), *user interface subsystem* (Harmony/UI), and *object-oriented database* (Harmony/DB). The function of these subsystems and an outline of their software architecture is described. The system configuration and its current implementation of the prototype system are discussed in section 6. The prototype system of *Harmony* is running on a SUN3/470[2] based on the UNIX operating system[3], and the motion video is handled by an *X window* system[4] using the Parallax graphic board installed in the SUN3/470. Through this development of the prototype system, we examine the effectiveness of our design methodology. In section 7, we make a brief review of the relevant works in hypermedia systems with link semantics. By comparing *Harmony* to these systems, the significance of our proposed hyperobject system is demonstrated. Section 8 concludes our discussion on *Harmony* and suggests several future works.

2 Features of "Harmony"

The features of *Harmony* are:

- integration of a hypertext model with an object-oriented framework,
- ability to handle temporal and active media,
- object-oriented database basis,
- *open* hypermedia system.

[1]VERSANT is a trademark of Versant Object Technology Corporation.
[2]SUN3/470 is a registered trademark of SUN microsystems, Inc.
[3]UNIX is a registered trademark of AT&T.
[4]The *X window* system is a registered trademark of MIT.

Integrating the hypertext model and the object-oriented framework

For treating very complex structures including the relations between and the manipulation of data in multimedia information systems, the conventional notion of nodes and links in hypertext is extended by the introduction of an object-oriented paradigm.

In the object-oriented framework, an object contains data as well as procedures (i.e., *methods*) to manipulate the data. In *Harmony*, multimedia data and their associating procedures are entirely encapsulated within an object. In addition, each object is considered as a node in the hypertext system, and the relations among objects are represented as links which connect the corresponding nodes. Thus, to construct a powerful model for a multimedia information system, the hypertext model and the object-oriented paradigm are integrated to utilize the advantages of both frameworks.

Handling temporal and active media

The integration of hypertext and object-oriented frameworks is insufficient to control information requiring real-time actions and the synchronization of operations among different media. For manipulating information with such complex relations, we have extended the ordinary notion of a link so as to describe the time constraints of each link. For the inclusion of motion video or music into hypertext models, time constraints on links become more important since both the information contained in the motion video and the music change continuously. Therefore, it is necessary to add a link from/to the information at a specific time. The ordinary notion of hypertext does not consider this kind of link because it cannot deal with any conditional branches specifying time constraints on a link. We have extended the notion of a link to handle these kinds of conditions and information.

Based on the object-oriented database

As the number of different media involved in *Harmony* increases and as their relations become more complicated, the object-oriented database is more suitable for storing information with complex data structures as compared with other database models such as the relational database. In *Harmony* as described above, data and their associated procedures are entirely encapsulated within an object. For this reason, it is natural to employ an object-oriented database. We have employed commercially available object-oriented database VERSANT which is suitable for hypermedia systems. Note that the decision for introducing the VERSANT system is due to the following simple reasons:

1. The base programming language of *Harmony* is C++ and VERSANT has a C++ interface.

2. Of the commercially available object-oriented databases based on the C++ language, VERSANT was the first and most easily available to our laboratory at that stage of system development.

Developing an open hypermedia system

The system should not depend on or be limited to the currently available media; it must be sufficiently flexible to include new media. In this sense, the hypermedia must be *open* for new media. In general, the object-oriented framework is very helpful for the construction of flexible systems because it strengthens the independence of each component.

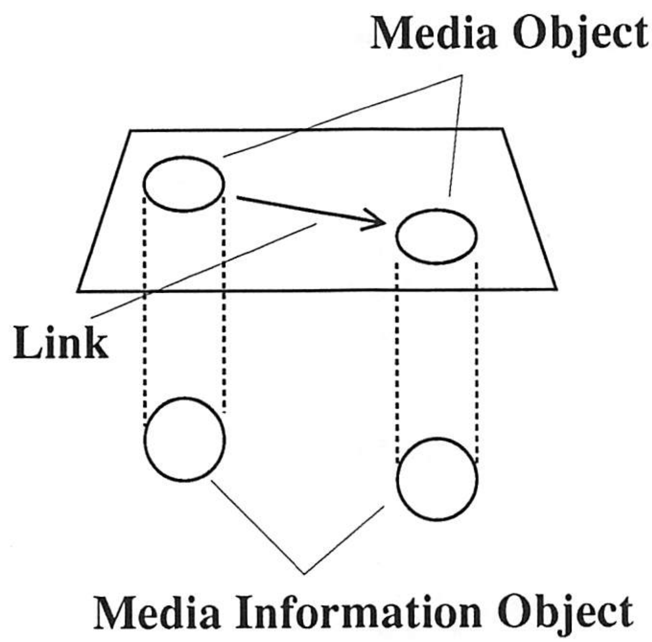


Figure 1: Hyperobject model

Because of the value of this property, our hyperobject system *Harmony* was designed as an open system to include new attractive media.

3 Hyperobject Model

In this section, we discuss the *hyperobject* model which is the basic modeling methodology for *Harmony*.

In *Harmony*, each medium has its own internal data structure and procedures to treat its information. Because a piece of information of a medium is represented as an object, data is represented as the status of the object, and procedures are represented as the methods of the object. Objects are connected by links in hyperobject systems, just as nodes are in hypermedia systems. However, our notion of objects and links is slightly different from that of the ordinary hypertext system.

The description of links connecting objects is separate from that of the objects themselves. This description is called a *layer*. Therefore, a hypermedia document consists of a link description (i.e., layer) and a set of objects in the layer.

All objects involved in various media on a layer are treated in the same way. To achieve such a property, a restricted and common interface to objects of a media was provided on a layer. This restricted object is called a *media object* (see Figure 1).

3.1 Link Model

Since an object is considered as a *node* in hypertext systems, a link between nodes represents a relation between objects. In our link model, when a user navigates from one node to another, a message is sent from the source node to its destination node through the link between them.

The notion of a link in *Harmony* consists of *objects* connected by the link, *conditions* denoting when the link is navigated, and *messages* sent to the target object. A link is represented by the following four-tuple:

```
<from, conditions, to, message>
```

The `from` field represents the source object of the message; the `to` field represents the destination object of the message. The `conditions` field describes the conditions when the link is activated. Finally, the `message` field specifies the *selector* (i.e., method name) and parameters of the message sent to the target object when the conditions are satisfied.

Using this mechanism in *Harmony*, one can specify temporal relations between objects, such as, a piece of music starting 30 seconds after a video begins, or the display of a particular piece of text starting when a video finishes. The link descriptions of these examples are given as follows:

```
<aVideo, started: 30, aMusic, play>
<aVideo, finished: 0, aText, play>
```

As shown above, a link specifies an object and the relations between the objects. Thus, the notion of a link in *Harmony* is more powerful than that in hypertext. One can also express temporal relations between objects, and control active media through links. For example, music can be stopped from a specified word in a text by navigating the link. If several links have their conditions satisfied at the same time, these links are navigated in parallel. Thus, we can describe a model having parallel execution.

3.2 Media Object

By representing an object as a node in a hypertext, we can extend the reusability of objects. For instance, an object can be used in multiple hypertext documents. Alternately, for connecting objects by a link, the interface between these objects must match easily. The interface among objects must be therefore as consistent as possible; it is desirable to provide a common interface among objects. We prepare the following interface to a *media object* which is shown in Table 1. The interface shown in the table is common to all *media objects*, however, the implementation of their methods depends on each media.

In the current release of *Harmony*, we can take five media as objects: text, music, graphics, motion video, and computer animation. All these objects have their own editing methods as well as associated management methods, however, these methods are not used in the link description.

3.3 Subobject

In usual hypertext systems, a part of a text such as a word or a paragraph can be specified as a source/destination of a link. Similarly, a character in a motion video or a

Table 1: The interface of a *media object*

Receive	Send
play stop	started: time finished: time clicked

rectangle in a graphic can be specified as a source/destination of a link. To fulfill these requirements in *Harmony*, a part of an object can be also considered as an object. These kinds of objects are called *subobjects*. If *object A* is a subobject of *object B*, we shall call *object B* a *parent object* of *object A*. A subobject can send/receive messages as in Table 1 in the same manner of its parent. Although a subobject is treated in the same manner as the ordinary object, the definition of a subobject is dependent on its parent object. A subobject of a video object is defined as a rectangle area which has a specified location and a specified time duration relative to the parent object. While a subobject of a graphic object is defined as any part of a picture such as a line, an arc, a rectangle, etc. Each object can define its subobject in any way except that a subobject must be completely included in the parent object. The notion of a subobject is an object-oriented extension as well as generalization of *block* in Intermedia[3]. However, in *Harmony* a subobject is treated as an active and temporal object and has the same capability as that of its parent object. A subobject sends a *started* (*finished*) event when it starts (stops) displaying media and it can receive messages from other objects. This notion of subobjects extends the flexibility of our hyperobject model.

4 Temporal and Active Media

In this section, we discuss more about how to deal with *temporal* and *active media*.

4.1 Synchronization among Objects

If there are several links whose conditions are satisfied at the same time, these links are navigated in parallel. Thus, we can describe a model having parallel execution. To realize parallel execution, the synchronization among objects invoked by a parallel execution should be implemented. Suppose for example that three information media are displayed at the same time and we want to stop displaying these media at the same time. In this case, the earliest *finished* event in these three objects should be captured by the system for synchronization. To implement the synchronization mechanism, we introduce the notion of an *object group* in our link model. In the current implementation, we considered the following two types of semantics for the object group, where A, B, C, ... are members of a object group.

min(A, B, C, ...) : represents the earliest occurrence of an event in an object group.

max(A, B, C, ...) : represents the latest occurrence of an event in an object group.

These semantics can be used as a source of a link. Furthermore, in the case that an object group can become a destination of the link, a message is broadcasted to all members of the object group when the link is navigated. As an extension of an object group, the answer for a query can be given as an object group, which enriches the capability of a link description.

4.2 Temporal Relation in Link Model

To include temporal and active media in the hypertext system, we have to define the *temporal relation* among media such as *display ordering*. To represent temporal relations, we employed an *event based approach* rather than a *time-line based approach* as in many existing authoring systems[6, 7]. The time-line based approach is effective and easy when the relation among media is sequentially ordered from the beginning to the end. However, it is powerless to define more complicated relations such as the synchronization of multiple information.

As an example of the event based approach in *Harmony*, let us show how to specify the temporal relation that a piece of music starts 30 seconds after a video finishes. The link description for this example is given as follows:

```
<aVideo, finished: 30, aMusic, play>
```

Thus, by extending the specification of conditions incident to the link, *Harmony* can provide a mechanism to describe temporal relations among objects.

4.3 Implementation of Timing Control among Objects

In the example given in the previous subsection, the temporal relation between two objects is controlled externally based on the global system clock implemented in the link manager of the system (i.e., Harmony/LM) (see Figure 2). More specifically, assume that it is described that a piece of music starts 30 seconds after a video finishes as the above example. Then, Harmony/LM sends a *start* message to the music object 30 seconds after receiving a *finished* event from the video object.

However, we can consider another timing control based on the internal time. Generally, each *medium object* has its own definition of time such as a frame number in a video and a key frame in an animation. The timing control based on such internal time depending on each object should be performed by each object. For example, consider the case that a music object is described to start 10 seconds after a video object starts (note that in the previous example, 30 seconds are counted after the *finish* of the video. Thus, the internal time of video object is not concerned with this temporal relation). This description is given as the following statement:

```
<aVideo, started: 10, aMusic, play>
```

This 10 seconds should be measured by the video in the frame number because the video can change the playback speed and 10 seconds measured by a global clock becomes meaningless. This internal time control mechanism is implemented by the notion of a subobject as discussed above. In this example, before an execution of a scenario, a *pseudo* video subobject is defined by the Harmony/LM incident to the frame number which corresponds to the time 10 seconds after the start of the video. At the time 10 seconds (measured by the frame number) after the video starts, this pseudo subobject is invoked and a *started*

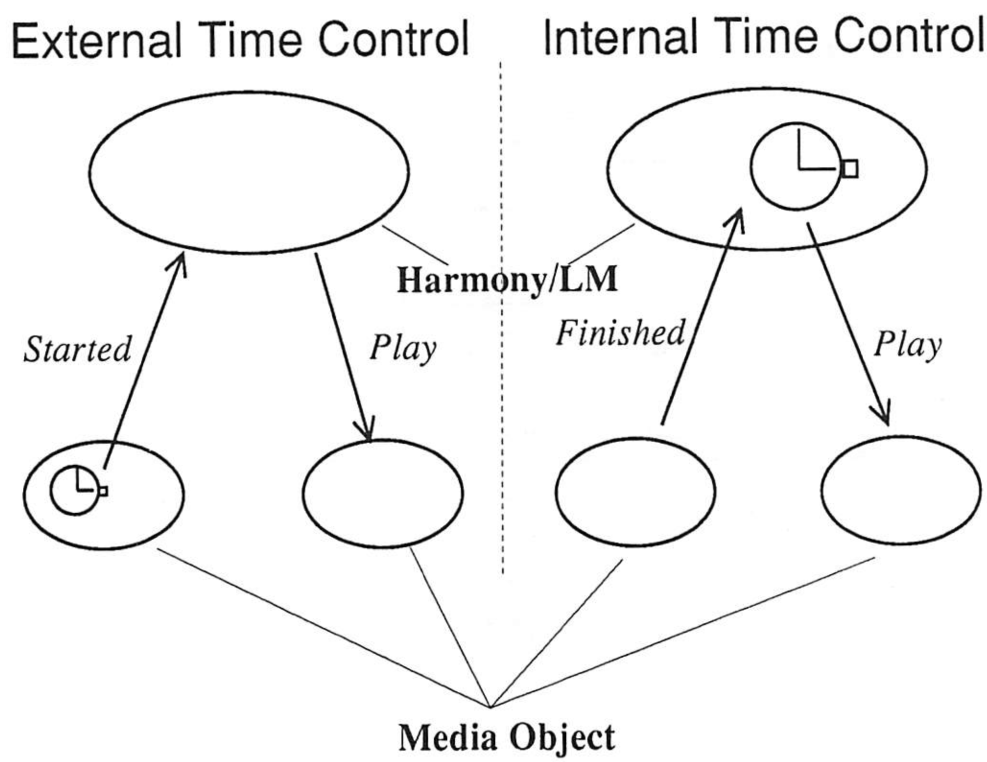


Figure 2: Two types of timing control.

event is sent to the Harmony/LM. On receiving the event, the Harmony/LM sends a *start* message to the music object.

However, media objects without any internal time mechanism such as text and graphics need not discriminate the global time from the internal one. Regarding these media, time is measured and controlled by the Harmony/LM.

5 System Architecture

In this section, the system architecture of *Harmony* is presented. It consists of three subsystems: the *link manager subsystem* (Harmony/LM), the *user interface subsystem* (Harmony/UI), and the *object-oriented database subsystem* (Harmony/DB) (see Figure 3).

5.1 System Overview

Harmony consists of three subsystems: the Harmony/LM, the Harmony/UI, and the Harmony/DB. *Media objects* and their associated methods are stored in the Harmony/DB. Each media object method is executed on a dedicated process.

To describe a hyperobject document, a user searches for an object from the Harmony/DB and creates a link to another object in the Harmony/UI. Links are managed by the Harmony/LM. When a user navigates a link, a message is sent from the source media object to the Harmony/LM. The Harmony/LM searches for a link which matches

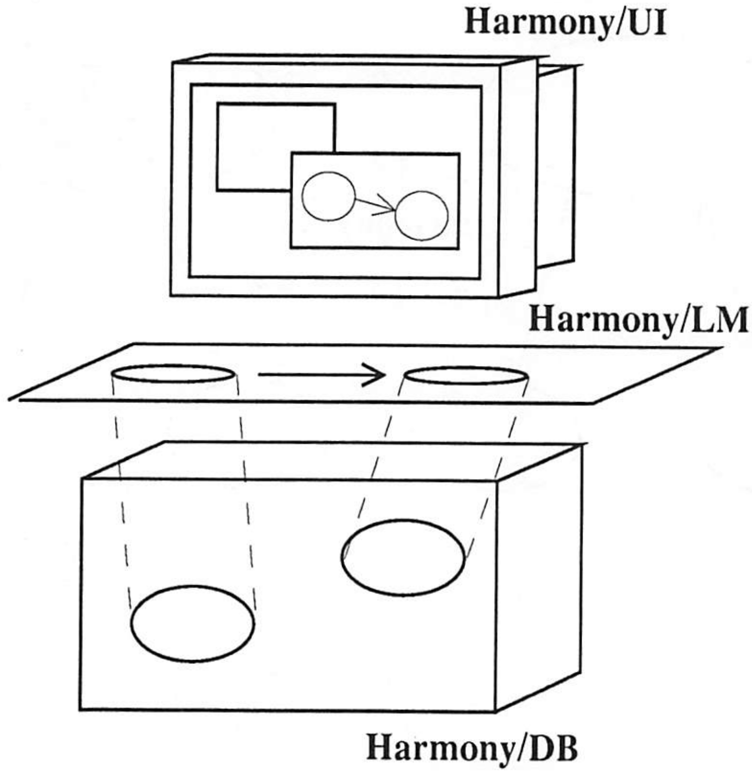


Figure 3: System architecture of *Harmony*

the message. If the desired link exists and its condition is satisfied, a specified message in the link is sent to the target object.

5.2 Harmony/UI

The *Harmony user interface subsystem* (Harmony/UI) provides a manipulating environment for objects using the support of a window system and a hardware system such as a mouse and a bitmapped display. In the Harmony/UI, one can create/navigate a link and search/retrieve an object in the integrated environment (See Figure 4).

Events from a user such as mouse clicking and key typing are translated into messages by the Harmony/UI and the messages are sent to the Harmony/LM or the Harmony/DB. Events for the creation/deletion/modification of links are sent to the Harmony/LM while the corresponding events for objects are sent to the Harmony/DB.

Recently we have developed the facility to illustrate the structure of hypermedia documents. This improvement to the user interface resolves disorientation and cognitive overhead when authoring hypermedia documents with a complex structure. This new function of the Harmony/UI is called the *scenario viewer*, and it displays the structure of hypermedia documents in a tree graph. This graph illustrates the time ordering or relations among media information (See Figure 5).

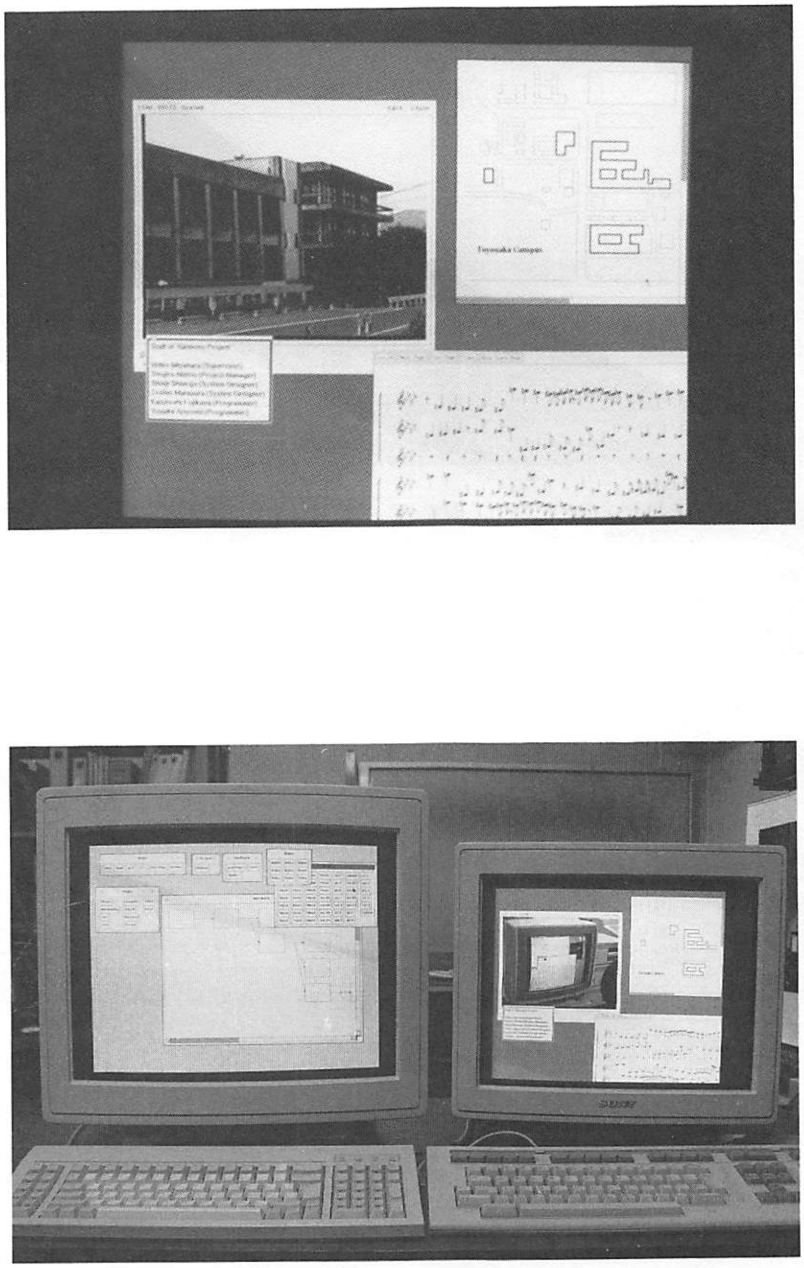


Figure 4: User Interface of *Harmony*

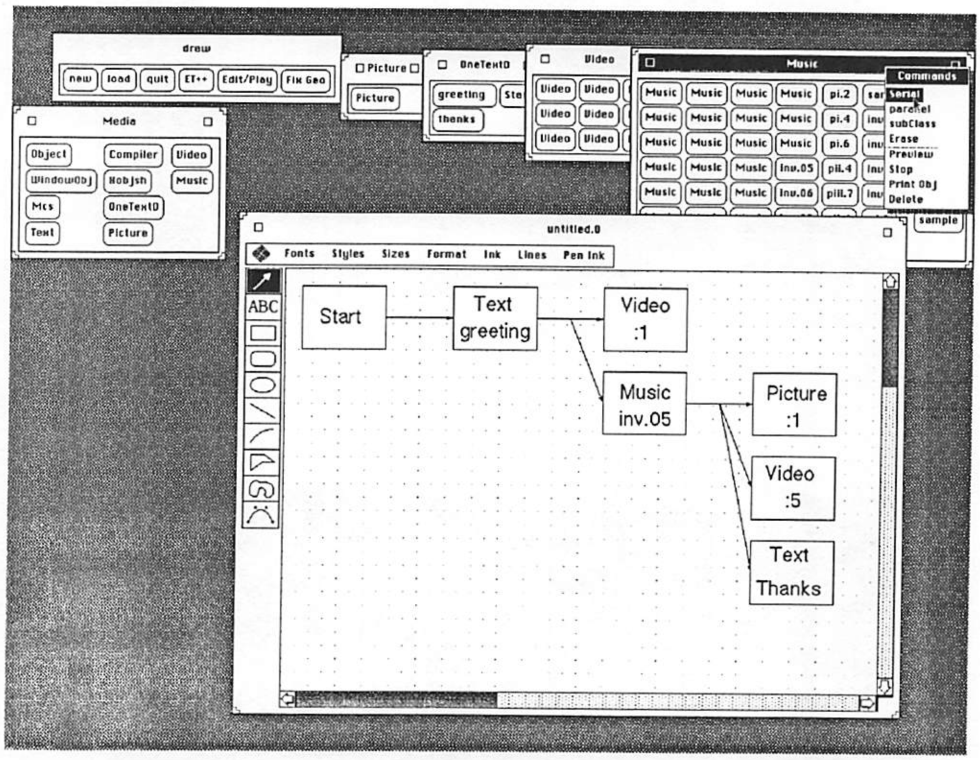


Figure 5: The scenario viewer

5.3 Harmony/LM

Links among a set of objects are managed by the *Harmony link manager subsystem* (Harmony/LM). A link is created in the Harmony/LM and stored in the Harmony/DB. When creating a hypermedia document, a message for adding/deleting links between objects is issued from the Harmony/UI. If a user intends to navigate a hypermedia document, a set of objects are retrieved from the Harmony/DB to the Harmony/LM. When it is necessary to navigate a link, a corresponding message including a source object identifier is sent to the Harmony/LM. On receiving the message, the Harmony/LM searches for a link which matches the message. If there exists a link whose conditions are satisfied, the specified message in the link is sent to the target object. This message is directly sent to the Harmony/DB and forwarded to the corresponding object.

5.4 Harmony/DB

To manage the complicated information and methods of *media objects*, the commercially available object-oriented database VERSANT was employed. This database has been developed using the C++ language and supports an almost full set of functions which are required for an object-oriented database[8], for instance, *object identity*, *complex object*, *encapsulation*, and an *inheritance mechanism*.

6 Current Implementation of "Harmony"

The Harmony/UI, the Harmony/LM, and the Harmony/DB are implemented together as a server process in UNIX. This process communicates with other media objects via sockets, which are an interprocess communications facility in the UNIX 4.3 BSD operating system. The Harmony/UI utilizes the *X window* system and ET++[5][9] to provide an efficient user interface as well as a direct manipulation environment for objects (see Figure 6). *Media objects* are also implemented as UNIX processes.

All processes run on a SUN3/470 with a *Parallax graphic board* which enables the handling of motion video on the *X window* system. Music objects run on a NeXT[6] computer with a *MIDI* interface and a *synthesizer*. These two computers are connected via a LAN and can communicate by means of the socket interface.

In *Harmony* presented in [5], the system was designed based on the object-oriented framework, but its implementation was not done completely according to object-oriented technology. Therefore, in [5] we pointed out several areas which require improvement:

- strength of Harmony/DB functionality,
- developing more advanced Harmony/UI,
- uniformity of employed programming languages,
- handling of subobjects.

In the following we will discuss the above problems in more detail, and present our resolutions which are implemented in the current systems.

[5]Toolkit ET++ is a C++ class library developed by UNILAB in The Union Bank of Switzerland.
[6]NeXT is a registered trademark of NeXT, Inc.

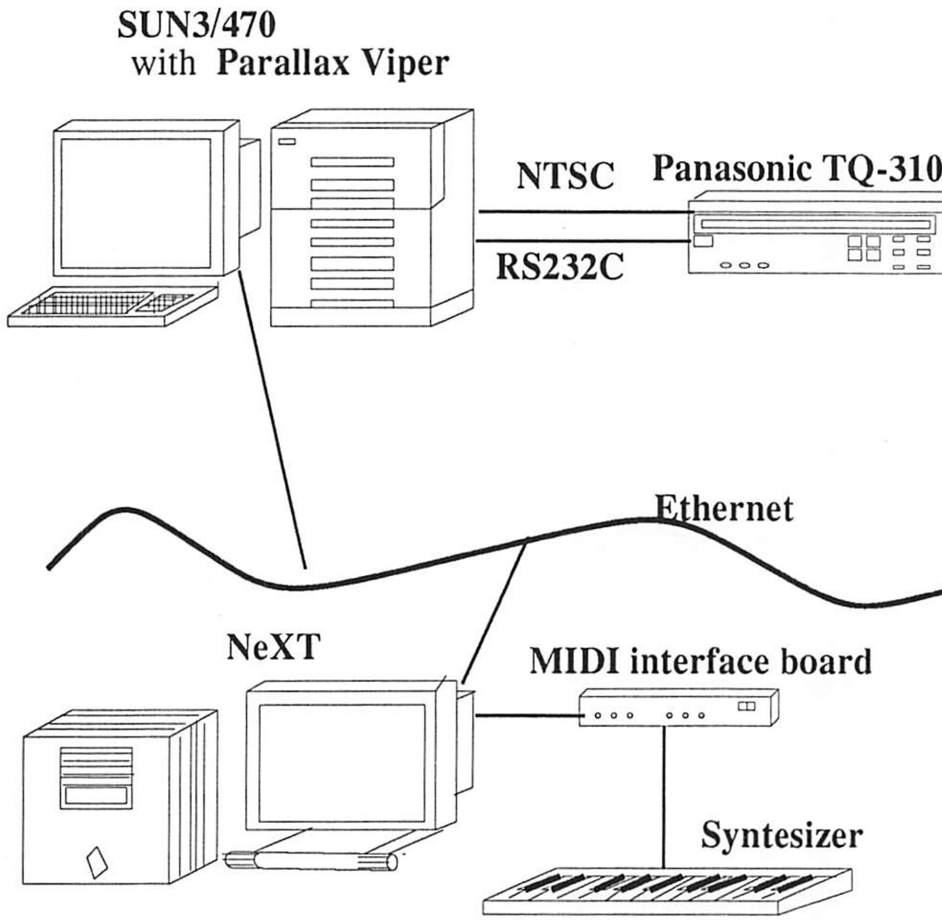


Figure 6: Hardware configuration of *Harmony*

6.1 Database

In the previous version of *Harmony*, the Harmony/DB which stores media information was developed using simple database *ndbm* on UNIX. However, the *ndbm* library is too simple to store and manage complicated media information, and the whole system becomes very complex for dealing with multiple media information. To improve the functionality of Harmony/DB, we introduced the commercial object-oriented database VERSANT as was discussed above.

Usually, object-oriented databases have the ability to model, store, and manage complex information directly. In addition to these capabilities, they can manage methods to deal with complex information. Such abilities of object-oriented database systems can greatly reduce the complexity of programming the whole system. Furthermore, since they adopt the *server/client architecture*, they can become a basis for distributed multiprocess applications such as *Harmony* where each process shares and exchanges information through the database server.

Although many commercial object-oriented database systems are now available, we found that they generally lack several capabilities for providing full-fledged functions required for the Harmony/DB. Thus, we developed the interface software on top of the employed object-oriented database for realizing the server/client architecture and the object identify property (see [10] for detailed discussion).

6.2 User Interface

In the previous version of *Harmony*, the Harmony/UI utilized the *X window* system and its *widget* mechanism to provide a user interface as well as a direct manipulation environment for objects. We used ATHENA widget to realize the complicated user interface component, however, ATHENA widget is too small and simple to provide a user interface to multimedia systems. Widget and Intrinsics[11] mechanisms provide an object-oriented framework without employing object-oriented languages. However, their objected-oriented framework is enclosed in the window system and is not applicable to building other system components. Therefore, to satisfy our requirement for the user interface, we employed toolkit ET++ which is a C++ class library. ET++ provides a consistent object-oriented framework for both of interface building blocks and applications. In addition to this feature, the other important features of ET++ are given as follows:

- The same binaries can run under several window systems such as *X window* system, NeWS, and SunWindows[7].
- It provides *meta-level* information such as the class structure, which is very useful for program debugging and for object-oriented database systems.
- It provides a new concept which can deal with a group of objects such as collection, set, and cluster. This concept is popular in database systems but is a new one for programming languages. Especially, it provides a very useful environment to build the user interface on window systems.

Considering such advantages, we updated the Harmony/UI and a couple of media processes via ET++.

[7]NeWS and SunWindows are trademarks of SUN microsystems, Inc.

6.3 Language

The previous version of *Harmony* was mainly developed using the C language. *Harmony's* development provides an example that an object-oriented system can be built upon without using object-oriented programming languages. However, the system became remarkably complex because it consists of many complicated components including the window system, the database system, and media presentation components. If we add further functions or media into *Harmony*, the system may become unmanageable. To avoid such a serious situation, we decided to employ object-oriented programming using C++ to implement the system, and recently the whole system has been updated. The C++ Language provides a sophisticated interface to the Harmony/DB as well as the Harmony/UI. Furthermore, using C++, both the encapsulated part and the interface part of each media presentation component can be implemented consistently within the object-oriented framework.

6.4 Subobject

A subobject is stored in the Harmony/DB as a separate object from its parent object. Thus, every parent object can be shared among users without any dependency of its subobjects, and furthermore the set of subobjects for a parent object can be personally defined according to the users.

To implement such independency of a object from its subobjects, the subobjects are loaded into its parent object just before a scenario starts. At the beginning of execution of a scenario, Harmony/LM sends a list of subobjects used in the scenario by the following message:

```
standby: a list of subobjects
```

On receiving this message, the object starts to load the subobjects from Harmony/DB.

7 Comparison with Related Works

In [5], to make the significance of our proposed system *Harmony* more visible, we made a brief review of the relevant works in hypermedia systems such as KMS[1], Intermedia[3], ATHENA Muse[6], Elastic Charles[7], Scripted Documents[12] and Trellis[13], and compared them with *Harmony*. In this section, by comparing *Harmony* with hypertext and hypermedia systems which include link semantics, we evaluate our system and discuss its advantages.

Navigation semantics on links in the hypertext document[13, 12, 14] is introduced to eliminate user disorientation and high cognitive overhead[15]. In the Trellis system[13], a Petri net[16] is introduced to describe link semantics on hypertext document. Prerequiry or users' intention in navigating a document can be described in the Petri net. The navigation semantics forms a *bipartite graph* structure represented as a Petri net model. The basic system elements in this Petri net model are *contents*, *window*, and *button* where *button* represents a transition of *contents*. However, *button* cannot include multiple semantics and only corresponds to a single action in the system. On the other hand, events in *Harmony* can represent multiple semantics. The current system Trellis can not deal with temporal and active media, and the underlying Petri net model must be extended for including temporal relations among information media.

In [12], the *path* mechanism is introduced on top of the hypertext document. Zellweger introduced the notion of an *active entry* which can perform actions such as playing animation of pictures or playing back voice. An *active entry* is similar to the notion of *object* in *Harmony*, but it does not produce an event in the middle of its action. In *Harmony*, an object can produce events during its action and an event can become a source of a link. The path is represented by a *script* in a document and has no graphical representation.

According to the above observation, we can say that the link semantics of *Harmony* is very advanced for dealing with temporal and active media.

8 Conclusion

In this paper, we presented a design methodology for hypermedia systems including temporal and active media, such as music sounds, motion video, and animation based on the object-oriented framework. Our link model is based on *hyperobject* which integrates the object-oriented framework and hypermedia model. By the application of our model, we have developed a prototype hyperobject system with text, music, graphics, motion video, and animation as objects. We have shown the effectiveness/advantages of our system model. The prototype system works well and supports parallel execution in a primitive distributed system environment. We evaluated *Harmony* by comparing it with several well known hypertext and hypermedia systems with link semantics. As we are just standing at the gate of a multimedia era, various matters including the ones listed below are left for our future research.

Real-time capability

Since a *media object* in *Harmony* is implemented as a process on the UNIX operating system, a tight synchronization among multiple media is very hard to realize. For example, we cannot guarantee that a music object and a video object start at the same time, although we can define such a link description. To satisfy very tight time constraints, operating systems with real-time capability or special hardware devices for synchronization will be required.

Extension to a distributed environment

The current version of *Harmony* has a network extensible object identifier, and it is an example of applications of distributed real-time systems. In order to develop such a system completely, the description mechanism of the global ordering of messages among multiple processes is important and has to be implemented.

Acknowledgments: The authors would like to express our appreciation to Mr. R.M. Sasnett of Project ATHENA of MIT for providing the *X server* running with the *Parallax graphic board* and his technical support. Thanks are due to Mr. M. Kajimoto, Mr. Y. Ariyoshi, and Mr. M. S. Tsubata of Osaka University for their excellent programming work in assisting in the completion of our prototype system of *Harmony*. We wish to acknowledge Mr. E. Miska of University of Victoria for his valuable comments on an earlier version of this paper. We also thank CANON INC. for providing us the environment to use a NeXT computer.

References

[1] Akscyn,R.M., MacCracken,D.L., and yoder,E.A. : *"KMS: a Distributed Hypermedia System for Managing Knowledge in Organizations"*, Communications of the ACM, vol.31, no.7, pp.820-835, July 1988.

[2] Halasz,F.G. : *"Reflections on NoteCards: Seven Issues For The Next Generation of HyperMedia Systems"*, Communications of The ACM, vol.31, no.7, pp.820-835, Jul. 1988.

[3] Yankelovich,N., Haan,B.J., Meyrowitz,N.K., and Drucker,S.M. : *"Intermedia: The Concept and the Construction of Seamless Information Environment"*, IEEE Computer, vol.21, no.1, Jan. 1988.

[4] Goldberg,A., and Robson,D. : *"Smalltalk-80: The Language and its Implementation"*, Addison Wesley, 1983.

[5] Shimojo,S, Matsuura,T., Fujikawa,K., Nishio,S., and Miyahara,H : *"A New Hyperobject System Harmony: Its Design and Implementation"*, International Conference on Multimedia Information Systems, McGraw-Hill, pp.243-257, Jan. 1991.

[6] Hodges,M.E., Sasnett,R.K., and Ackerman,M.S. : *"A Construction Set for Multimedia Applications"*, IEEE Software, vol.6, no.1, pp.37-43, 1989.

[7] Brøndmo,H.P., and Davenport,G. : *" Creating and Viewing the Elastic Charles - a Hypermedia Journal"*, Proceedings of Hypertext 2, June. 1989.

[8] Atkinson,M., Bancilhon,F., DeWitt,D., Dittrich,K., Maier,D., and Zdonik,S. : *"The Object-Oriented Database System Manifesto"*, in Deductive and Object-Oriented Databases, Kim,W., Nicolas,J.M., and Nishio,S., Eds. Amsterdam : North-Holland, pp.223-240, 1990.

[9] Weinand,A., Gamma,E., and Marty,R. : *"Design and Implementation of ET++, a Seamless Object-Oriented Application Framework"*, Structured Programming, vol.10, no.2, June 1989.

[10] Shimojo,S, Matsuura,T., Fujikawa,K., Nishio,S., and Miyahara,H : *"Architectural Issues on Multimedia Presentation System Harmony"*, Dept. of Information and Computer Sciences, Osaka University, Working Paper, Mar. 1991.

[11] Swick,R.R., and Weissman,T. : *"X Toolkit Athena Widgets - C Language Interface"*, in X Window System Version 11, Release 3. Massachusetts Institute of Technology.

[12] Zellweger,P.T. : *"Scripted Documents: A Hypermedia Path Mechanism"*, Proceedings of Hypertext 89, pp.1-14, Nov. 1989.

[13] Furuta,R., and Stotts,P.D. : *"Programmable Browsing Semantics in Trellis"*, Hypertext 89, pp.27-42, Nov. 1989.

[14] Marshall,C.C., and Irish,P.M. : *"Guided Tours and On-Line Presentations: How Authors Make Existing Hypertext Intelligible for Readers"*, Proceedings of Hypertext 89, pp.15-26, Nov. 1989.

[15] Conklin,J. : *"Hypertext: An introduction and survey"*, IEEE Computer, vol.20, no.9, pp.17-41, Nov. 1987.

[16] Peterson,J.L., Petri Net Theory and the Modeling of Systems. Englewood Cliffs, NJ: Prentice-Hall, 1981.

Kazutoshi Fujikawa received the M.E. degree from Osaka University in 1990. He is now a Ph.D candidate in the Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University. His current research interests include hypermedia systems and distributed systems.

Shinji Shimojo received the M.E. and D.E. degrees from Osaka University in 1983 and 1986, respectively. From 1986 to 1989, he was an Assistant Professor in the Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University. Since 1989, he has been an Associate Professor of Computation Center, Osaka University. He was engaged in the project of object-oriented network environment and shared widget on X window system. His current research interests include user interface of multimedia systems, distributed systems, and object-oriented database systems.

Toshio Matsuura received the M.E. degrees from Osaka University in 1977. Since 1979, he has been an Assistant Professor in the Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University. He is a designer of *key3*, a drawing tool on X window system. His current research interests include programming environment and user interface.

Shojiro Nishio received the B.E., M.E., and Dr.E. degrees from Kyoto University, Kyoto, Japan, in 1975, 1977, and 1980, respectively. From 1980 to 1988, he was with the Department of Applied Mathematics and Physics, Faculty of Engineering, Kyoto University. Since 1988, he has been with Osaka University, Toyonaka, Osaka, Japan, where we is an Associate Professor in the Department of Information and Computer Sciences and the Education Center for Information Processing. He has held visiting appointments with the University of Waterloo and the University of Victoria in Canada. His current research interests include database systems, knowledge-base systems, and distributed systems. He currently serves on the editorial boards of *Data and Knowledge Engineering* and *New Generation Computing*.

Hideo Miyahara received the M.E. and a D.E. degrees from Osaka University in 1969 and 1973, respectively. From 1973 to 1980 he was an Assistant Professor in the Department of Applied Mathematics and Physics, Faculty of Engineering, Kyoto University. Since 1980, he has been with Osaka University, Toyonaka, Osaka, Japan, where he is a Professor in the Department of Information and Computer Sciences. From 1983 to 1984, he was a Visiting Scientist of IBM Thomas J. Watson Research Center. His current research interests include performance evaluation of computer communication networks, local area networks and distributed systems.

Spatiotemporal Editing for HDTV Program Production

Seiki Inoue *Masahiro Shibata*

Science and Technical Research Laboratories,
NHK(Japan Broadcasting Corporation)

Abstract

We have been developing a component image database for scene synthesis. This report describes a new scene generation method using that image database and a multi-layered image synthesizer. In this method, several component images transferred from the database are arranged on multiple layers in spatiotemporal domain and combined to a scene or a whole program itself. We call this method "spatiotemporal editing". An experimental HDTV(Hi-Vision) system was also developed and usefulness of this method was confirmed through actual Hi-Vision program productions.

1. Introduction

Recently advanced broadcasting hardware has helped producing diversified media, such as satellite broadcasting and CATV, while upgrading the picture quality of these media, as represented by HDTV(Hi-Vision). Development of hardware has also created innovative image synthesizing techniques as seen in Synthevision(1) and the video mat method(2). Meanwhile, frequent use of high-quality images synthesized with these advanced techniques is now a major cause of rising TV program production costs. Therefore broadcasters have an ardent desire to find some adequately cost-effective method to produce high-quality images conforming with various broadcasting media.

What is currently envisaged is a system which can manipulate images at will on a work station. Such a system, however, may need as a prerequisite the developing of not only high-speed-operation hardware but a high-performance image synthesizing, editing, and storing method as well.

We have proposed a scene synthesizing method using a component image database(3). In this report, we will discuss a new image producing method which features an innovative idea of spatiotemporal editing evolved from conventional scene synthesizing techniques. This method simultaneously carries out image synthesis and editing by using images spatially and temporally arranged with a multi-layered image synthesizer. In the following sections, we will discuss the spatiotemporal editing method and image production processes which implement this method. Also we will examine if this method works as conceptualized using an experimental Hi-Vision system we have assembled.

2. Spatiotemporal editing

In our conventional image production terminology, spatially overlapping an image on another image is called "synthesis" and temporally connecting an image with another, "editing". In other words, the synthesis here means mixing a background image with a foreground image using the foreground image's key information. Chroma key synthesis produces key information from the blue back's color information, while the video mat

method manually inputs key information using the painting function of computer graphics. When synthesizing several images, the image stored in the farthest depth comes first. Since carrying out synthesis in such a fashion requires accurate timing, the operator needs a detailed time chart and if he failed in the middle of a synthesizing process, he would have to repeat the whole process from its very beginning. To solve this problem, it is desirable that several images arranged in the order of depth can be synthesized at the same time. In the computer graphics field, a method has been proposed for simultaneously synthesizing several objects drawn beforehand with respectively different display methods[(4)]. This method retains information about the object's picture element area (pixel coverage) as an α channel for synthesis.

Editing. A simple editing job connects images with a cut method but special editing methods, such as wipe and dissolving change, are tantamount to synthesizing two images using key information. These methods are used to change one image to another by temporally altering key information. They somewhat differ from synthesis in that any change of key information goes independently from the form of objects in an image. Change in the timing of key information (scene change timing), however, relates with image content and calls for adequate case-by-case adjustment.

As discussed above, there is no substantial difference between synthesis and editing. Both processes are used for mixing several images using key information and need delicate timing adjustment in so doing. Therefore it may be possible to simultaneously execute ordinary synthesis and editing processes and compose specific scenes or a program as a whole by temporally and spatially positioning relevant images and pieces of key information and synthesizing them into an image output. We call this entire process "spatiotemporal editing".

3. New image production system using spatiotemporal editing method

3.1. System configuration

Figure 1 (a) shows the configuration of a new image producing system we propose. This system synthesizes images with so to speak a stage setting method by laying out material images (image plane) and key information (α plane) in a multi-layered spatiotemporal structure in accordance with the description of scenes stored in the scene database, as illustrated in Figure 1 (b). Each part of the system plays the following roles:

[1] Component image database

The idea of a component image database is detailed in the reference (3) listed at the end of this report. In short, it stores and manages material images in the form of image information and key information pairs.

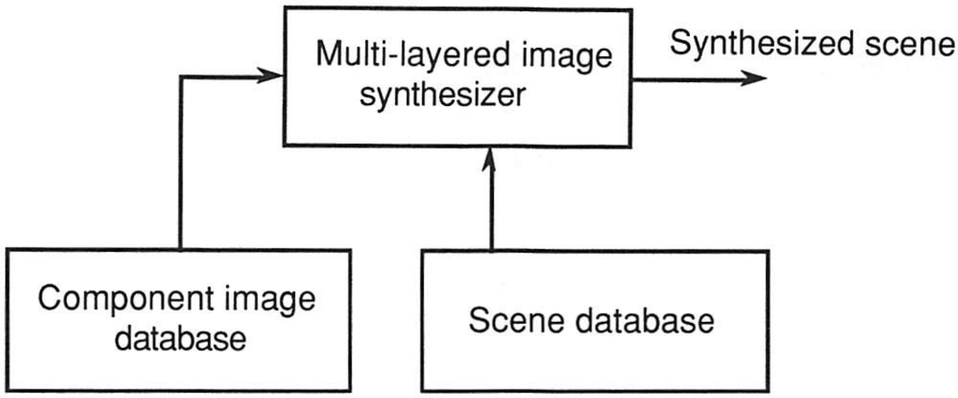
[2] Multi-layered image synthesizer

This synthesizer has a multi-layered frame memory system which simultaneously synthesizes images put into the layers (stage setting synthesis).

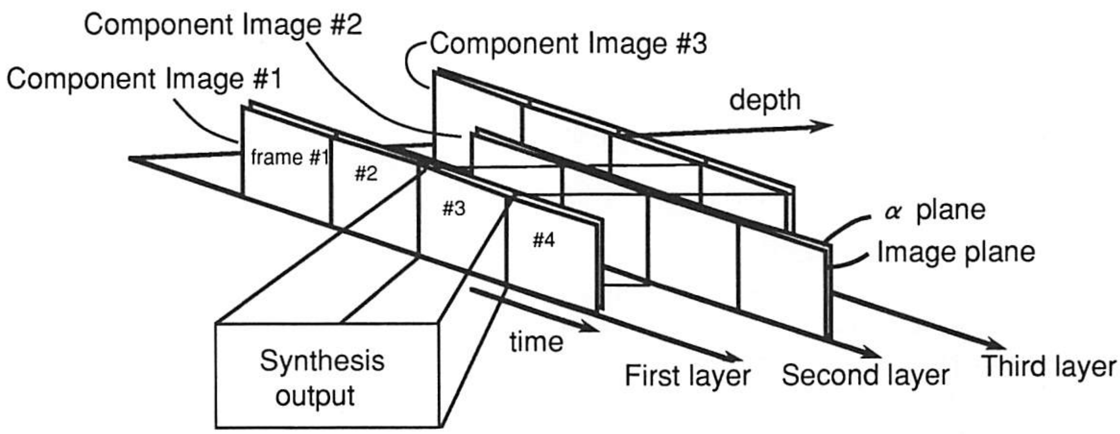
[3] Scene database

Each scene is defined by the spatiotemporal positions of component images

which may comprise the scene. A wipe or dissolving effect can be produced by using adequate forms of key information independent from images at the time of defining scenes.



(a) System configuration.



(b) Outline of spatiotemporal editing.

Fig.1 Spatiotemporal editing system.

3.2. Stage setting synthesis

Stage setting means flat backgrounds with natural scenery, such as trees and mountains, painted on them for use on the stage. Stage setting synthesis is a method of generating a quasi-three-dimensional effect by spatially laying out two-dimensional images in the depth direction. Each frame memory has image data (R, G, B) and opacity data ($\alpha : 0 \leq \alpha \leq 1$) for each picture element. When the foreground image is completely opaque ($\alpha = 1$), component images in the background are hidden behind it and invisible.

This entire relationship may be formulated as below, on condition that the number

of layered frame memories be represented by n, the image data in an arbitrary picture element of each layer at an arbitrary time, by f_i (i = 1 to n, f : R, G, or B data), opacity data, by α_i, and the layer depth from this side to the far side, by f_1, f_2, f_3, ... f_n, and that the image output f_o and the entire opacity output α_o be executed.

$$f_o = \alpha_1 f_1 + (1-\alpha_1)\,[\,\alpha_2 f_2 + (1-\alpha_2)\{\alpha_3 f_3 + \ \ldots + (1-\alpha_{n-1})\,\alpha_n f_n\}]$$

$$= \alpha_1 f_1 + \bar{\alpha}_1 \alpha_2 f_2 + \bar{\alpha}_1 \bar{\alpha}_2 \alpha_3 f_3 + \ldots + \bar{\alpha}_1 \bar{\alpha}_2 \bar{\alpha}_3 \ldots \bar{\alpha}_{n-1} \alpha_n f_n \qquad (1)$$

$$\text{where : } \bar{\alpha}_i \equiv 1 - \alpha_i$$

$$\alpha_o = 1 - \bar{\alpha}_1 \bar{\alpha}_2 \bar{\alpha}_3 \ldots \bar{\alpha}_n \qquad (2)$$

The above Formula (2) indicates that when at least a term from α_1 to α_n is 1 (completely opaque), the formula will represent $\alpha_o = 1$ (completely opaque).

The component image database stores material images in the form of f and α pairs. This enables synthesizing several component images and registers them as new component images. In other words, any component image can be inserted into any layer on the multi-layered image synthesizer. This dramatically augments image synthesis flexibility compared with conventional methods which require strictly sequential synthesis starting from the deepest picture.

In addition, opacity data attached to each picture element enables both translucent display and generation of soft key on the edge of component images.

4. Experimental system for Hi-Vision images

4.1. System configuration

To see if the above-mentioned new image producing system works well and efficiently, we have developed an experimental system for Hi-Vision images. Figure 2 shows the configuration of the experimental system and Figure 3 illustrates the configuration of an envisaged image synthesizer. Most of the synthesis material used here are still images.

For synthesis, component images (still images) are transferred from the component image database on a magnetic disk to a large-screen frame memory. Images in Hi-Vision size are cut out of the frame memory for stage setting image synthesis.

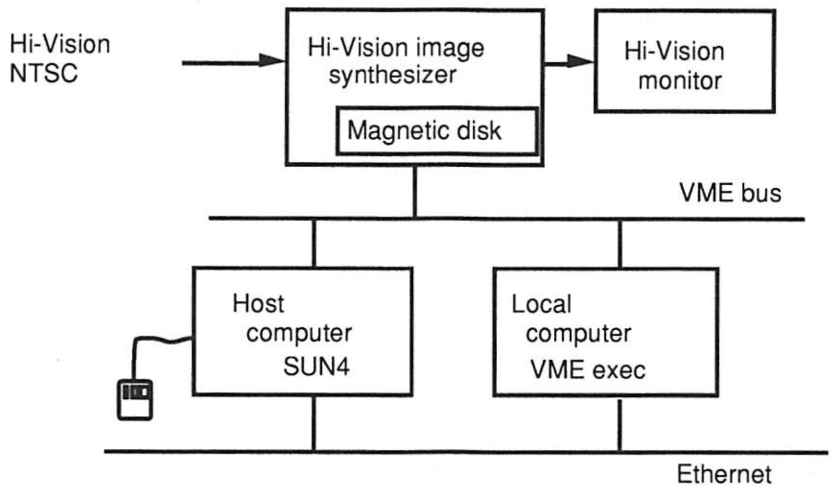


Fig.2 Configuration of the experimental system.

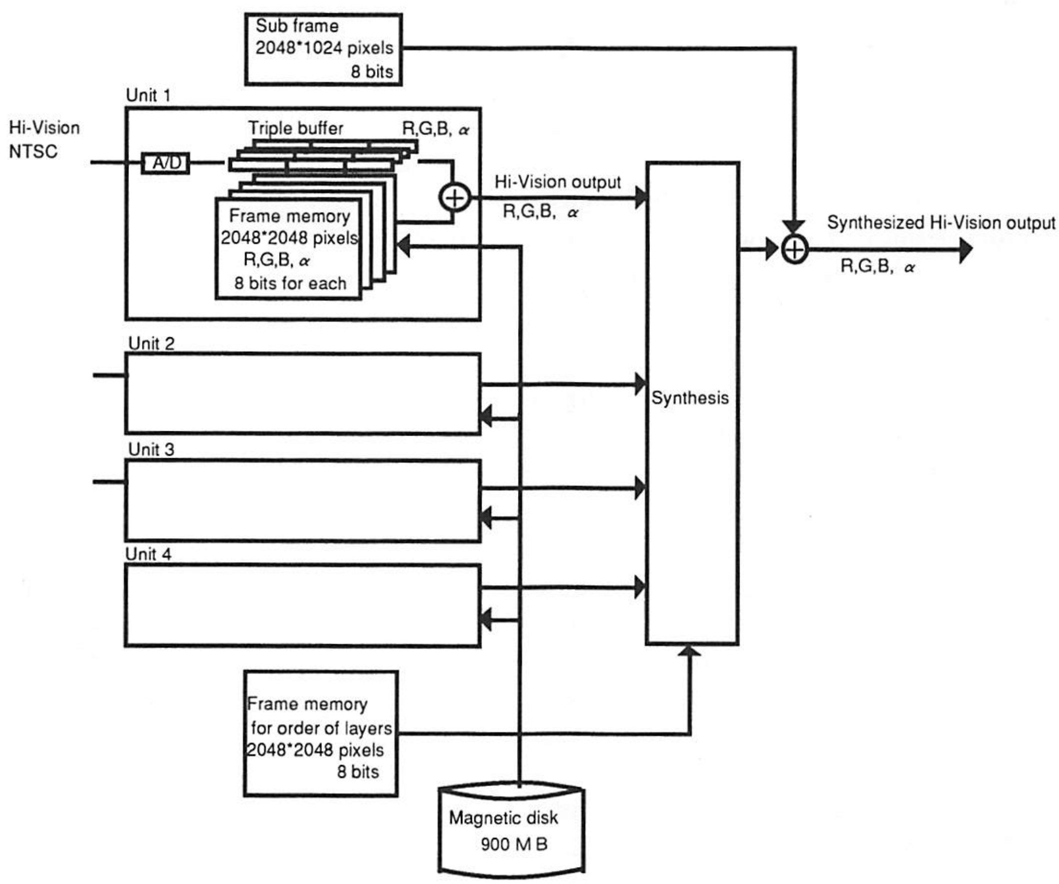


Fig.3 Configuration of the Hi-Vision image synthesizer.

• Component image database

The component image database, which is constructed on the magnetic disk of the Hi-Vision image synthesizer and managed by the host computer, stores still component images cut out in arbitrary sizes. The database stores not only these component images but also a variety of still image materials, including those larger than Hi-Vision size, input via scanner for use in synthesis.

• Image synthesizer

The image synthesizer has the following features:

[1] Equipped with 4-layer large-screen frame memories (2,048 X 2,048 pixels) larger than a Hi-Vision size (1,920 X 1,035 pixels).

[2] Each frame memory independently scrolls at real time (1/60 sec). In other words, Hi-Vision size images can be output from any portions of the large-screen frame memories.

[3] The large-screen frame memories not only receive component images from the database but also input movie images (Hi-Vision, NTSC) into any desired location in them in real-time.

[4] Each frame memory has a plane (α plane : 2,048 X 2,048 X 8 bits) for opacity data so as to execute stage setting synthesis in the depth order. The α plane can be scrolled independently.

[5] Produces a sensation of depth using motion parallax which is caused by moving each image at a speed in accordance with the distance from the viewer's visual point.

[6] In addition to the above-stated image frame memories, the system also has depth order defining memories for determining the depth order for each picture element and sub-frame memories for displaying menus including icons.

• Inputting scenes in motion

The image synthesizer has the following methods for generating moving scenes:

[1] Scrolling still component images

Scrolls 4-layer frame memories independently from each other and in real time (1/60 sec).

[2] Moving component images

For producing the motion of an object (e.g. flapping of a bird), the synthesizer lays out on the frame memories a series of still component images which may constitute the object's motion as moving component images, synthesizes them by each frame at real time, and displays them at any desired portion of the synthesized image. A frame memory can store 64 frames of a 256 X 256 pixel picture.

[3] Inputting movie image

The synthesizer synthesizes movie images input from external sources (Hi-Vision, NTSC) and displays them on a desired portion of the screen. The synthesizer uses opacity data prepared within it. It can also input in real time key signal outputs from an external chroma key source for use as opacity data.

Further, the synthesizer can easily generate a variety of scenes by changing

component images and/or depth information.

Control of all the above-mentioned processes is done by the host computer with a interactive method. The local computer can reproduce motion data composed on the host computer by synchronizing them using vertical synchronization signals or external VTR time codes.

• Scene change

The formula for synthesizing two images may be expressed from the above-mentioned Formula (1), as follows:

$$f_o = \alpha_1 f_1 + (1 - \alpha_1)\, \alpha_2 f_2 \qquad (3)$$

Adequately altering the α_1 and α_2 values enables scene changes. Some examples:

[1] Dissolve

With $\alpha_2 = 1$, α_1 is changed between 0 and 1.

[2] Wipe

With $\alpha_2 = 1$, the α_1 plane is independently scrolled (See Figure 4).

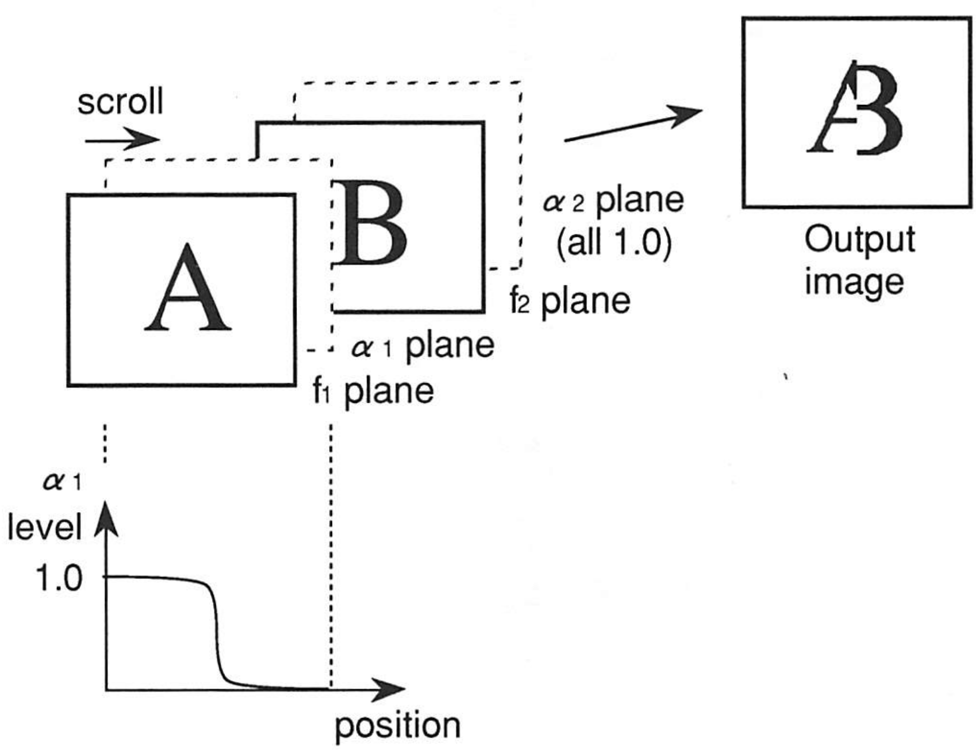


Fig.4 Wipe change.

4.2. Synthesizing example

Figure 5 shows a synthesized scene and Figure 6, a wipe change. The scene in Figure 5 has been composed by transferring the sky and a pyramid, the balloon, the bird, and the coconut tree, in that order, starting from the one (sky and pyramid) located deepest in 4-layer component image stocks. The bird image is a moving component image, while the other three are still components. Motion is generated by scrolled still component images and the moving component image. For composing the flapping of the bird, data for one flap, namely, the "bird" area, was cut out from a 26-field NTSC movie image and each of the field's data was interpolated into a frame data. Then the 26-frame data (128 X 128 pixels X 26 frames) was repeatedly displayed along the bird's flying locus thus creating a mock scene as if the bird was flying.

Fig.5 An example of a synthesized image.

Fig.6 An example of wipe change.

5. Conclusion

In the foregoing sections, we have proposed a new image producing concept which we call "spatiotemporal editing". We have shown that it is possible to implement this concept through efficient and flexible image production processes using a multi-layered image synthesizer which is combined with an image database. We have also proved the concept's feasibility with an experimental system we assembled for that purpose. This system has already been used for producing actual Hi-Vision programs and has been highly evaluated.

In order to upgrade the system's performance, we will improve and/or develop the following functions:

[1] Scrolling by a unit less than pixels

An accuracy in the order of a quarter of a picture element or under is said to be necessary for scrolling a picture so it may look natural to the human eye. We need to develop a highly accurate scrolling method using an interpolation filter.

[2] Enlarging/reducing functions

More attractive image synthesis may be possible if we can change the position of component images in their depth direction in real time. We will need real-time enlarging/shrinking functions.

[3] Synchronous VTR control

At present, VTRs are controlled manually. For spatiotemporal editing of images taken from several VTRs, it is necessary to control these VTRs so they may record or play back strictly to a reference time chart compiled for such editing.

[4] Efficient and simple scene data input

Not only the motion of each component image but the motion of one component image in relation to that of another is very important for producing programs with the spatiotemporal editing method. We will have to develop a method of easily inputting such scene motion data and storing them in a database for management.

We also have a plan to apply the image producing method proposed in this report to the production of video images taken from movie image material using an image database. For that purpose, we need a medium which can efficiently store image material including key information. Regrettably, however, VTRs we have today are unable to store key information together with image material. It is necessary for us to thoroughly study possible media and methods to store such information.

References

(1)Sigeru Shimoda, Masaki Hyashi, Yasuaki Kanatugu : "New Chroma-key Imaging Technique with Hi-Vision Background" , IEEE Transactions on Broadcasting, Vol.35, No.4, pp.357-361, December 1989.

(2)Akira Iwata, Yoshio Monjo, Teruo Niikura, Hisano Tamura : "A New Method of Video Synthesis Developed by NHK", SMPTE Journal, Vol.95, No.7, pp.702-704, July 1986.

(3)Seiki Inoue, Masahiro Shibata : "Component Image Database for Scene Synthesis", NHK Laboratories Note, Serial No.385, October 1990.

(4) Thomas Porter, Tom Duff : "Compositing Digital Images", Computer Graphics, Vol.18, No.3 , pp.253-259, July 1984, (acm SIGGRAPH '84 proceedings).

Seiki Inoue graduated in 1978 from the Electrical Engineering Department, Faculty of Engineering, Tokyo University, and obtained a Master's degree in Electronics from there in 1980. He joined NHK the same year. Since 1983, he has been with NHK Science and Technical Research Laboratories. He is engaged in research on image processing and image database.

Masahiro Shibata graduated in 1979 from the Electronics Department, Faculty of Engineering, Kyoto University, and obtained a Master's degree in Electronics from there in 1981. He joined NHK the same year, and has been assigned to NHK Science and Technical Research Laboratories since 1984. He is engaged in research on information retrieval technology and image database.

DIDDLY: Digital's Integrated Distributed Database LaboratorY

Ellen Lary
Database Systems Research
Digital Equipment Corporation

DIDDLY integrates multimedia heterogeneous databases and input/output that is audio, video, scanned text, keyboard or mouse. In this demonstration, we discuss the underlying architecture of DIDDLY and show the system in operation.

DIDDLY is a base platform and a set of standard services for manipulating multiple types of data. The base platform consists of a virtual network facility for communicating data and control information between cooperating processes, a control architecture, and a data access architecture. Application programmer interfaces (APIs) are provided for each of the base platform architectures. A standard set of services provide for capture and/or presentation of certain existing multimedia datatypes. The entire system is database driven and demonstrates the ability to provide a single interface to a network of heterogeneous database systems. All interfaces are built using industry standards such as SQL.

Using a patient management scenario, as well as other examples, we will show how DIDDLY integrates text, digitized audio, DECtalk Source Language, DDIF (CDA), IFF images, X window dumps, and full motion video into sophisticated applications. In addition, we will discuss data compression techniques that achieve up to a 60 to 1 compression ratio.

NEURAL ORCHESTRATION: FROM CORTICAL SIMULATION TO CORTICAL SYMPHONY

Matthew Witten *† AND Robert E. Wyatt ◇‡

Abstract. A computational model of signal flow in a vertically organized slab of neural tissue in cat area 17 has been developed. The modeled slab covers $500 \times 500\mu^2$ of cortical surface and extends vertically through the full depth of the cortex. The complexity of the data visualization requires both visual and audio. In this paper we deal with the unique issues of and the complex orchestral visualization of the cortical simulation. This leads us to discuss the concepts and formalizations necessary for a general theory of visualization

Key words. Neural Simulation, Neural Orchestration, BioSymphonics^{tm},Visualization, Symphonics, Realization, Representation, Actualization, Cognitive Descriptor, Visualization Descriptor

0.0. Introduction. The structure and the dynamics of the cerebral cortex continues to be a central theme for much ongoing experimental and theoretical neurobiological research. While a great deal is known about the physiological and synaptic properties of single neurons (Segev *et al.* (1989)), the dynamical behavior of systems of interacting neurons is much less accessible to direct experimental study and is thus still poorly understood. Increasingly, formal mathematical models and computer simulations make it possible to investigate the dynamical properties of systems of neurons. Realistic space-time simulations have been developed by numerous investigators (Sejnowski *et al.*(1989); Wilson and Bower(1989); Wehmeier *et al.*(1989); Traub et al.(1987, 1988, 1989)). Patton (1990,1991a,b) point out that the mammalian visual cortex is an attractive subject for computational modeling, since it has been the subject of an enormous body of experimental work. In addition, such work is of great theoretical interest.

Building mathematical models of biological/medical systems is both an art as well as a science. We do not know the real world dynamics $\mathbf{W}_R(\mathbb{S})$ of a particular system $\mathbb{S}$. At best, our *cognitive understanding* of the real world $\mathbb{S}$ is based upon our knowledge $\mathbf{W}_{KR}(\mathbb{S})$ of the system $\mathbb{S}$. This knowledge is gained by our observation $\mathcal{O}(t)$ of the real world system $\mathbb{S}$ through various experimental interventions. That is

$$\mathcal{O}(t) : \mathbf{W}_R(\mathbb{S}) \longrightarrow \mathbf{W}_{KR}(\mathbb{S})$$

We take this knowledge $\mathbf{W}_{KR}(\mathbb{S})$ and subsequently develop a cognitive understanding or model of $\mathbb{S}$. This cognitive understanding is then turned into one or more representations (models) $M_{\ast}$ of that understanding. It is important to be aware of the fact that representations of knowledge may be one-to-many in that a particular cognitive understanding may be mapped into numerous mathematical representations $M_{\ast}$, the set of which we denote by $\mathcal{M}$ and $M_{\ast} \in \mathcal{M}$.

A model $M_{\ast}$ is actually an n-tuple $M_{\ast} = \langle t, \mathcal{D} \rangle$ where $\mathcal{D} = \mathcal{D}_S \cup \mathcal{D}_D$, t is time, $\mathcal{D}_S$ is the set of *model descriptors* which are *static* in the representation (they do not change in time) and $\mathcal{D}_D$ is the set of *dynamic* model descriptors (they do change in time). We will discuss these in greater detail in a moment. For now, we shall define **visualization** as the act of transferring (transforming) our cognitive understanding of the real world and a subsequent representation $M_{\ast}$ to a sensoral realization $\mathcal{V}$. This realization $\mathcal{V}$, when actualized, is a visualization. Within this context, is clear that a visualization $\mathcal{V}$ is the action of a mapping $\mathcal{A}$ (the actualization mapping) such that the cognitive descriptors in the cognitive model are mapped into the visualization descriptors and subsequently actualized.

†Department of Applications Research and Development, University of Texas System, Center for High Performance Computing, BRC, 1.154CMS, 10100 Burnet Road, Austin, TX 78758 - 4497 USA
*E-mail: MWITTEN@CHPC.UTEXAS.EDU or MWITTEN@UTCHPC.BITNET
‡Department of Chemistry and Institute of Theoretical Chemistry, University of Texas at Austin, Austin, TX 78712 USA
◇ E-mail: CMAN041@HERMES.CHPC.UTEXAS.EDU

DEFINITION [1]. *Let* $\Delta = \{\delta_1, \delta_2, ..., \delta_l\}$ *be the set of cognitive descriptors in a given cognitive model. Let* Ψ *be the set of available visualization descriptors* $\Psi = \{\psi_1, \psi_2, ..., \psi_\alpha\}$ *. Then a visualization* $\mathcal{V}$ *is the actualization of the mapping*

$$\mathcal{A} : \Delta \longrightarrow \Psi$$

which may be a one-to-many mapping.

1.0. Mapping The Biology To A Simulation. The first stage of the problem is to map the neurobiology to an appropriate simulation. We begin our development by discussing the neurobiology and the method by which we have developed the simulation component of this project.

1.1. Modeling the Visual Cortex. The mammalian cerebral cortex is the most complex structure on Earth. It possesses a wide variety of cell classes interconnected in intricate ways. Cortical connectivity is both highly divergent in that many neurons synapse onto any given cell, and highly divergent in that each neuron contributes synaptic output to many other neurons. In addition, the signaling properties of the neurons themselves are highly non-linear. In such a situation, a knowledge of the anatomical connectivity of the system and the signaling properties of its neurons is far from sufficient to make the collective dynamical behavior and function of the cortical network intuitively obvious. Furthermore, the dynamical behavior of systems of interacting neurons is not readily accessible to direct experimental study.

In recent years computational modeling has emerged as a powerful tool for understanding complex systems of interacting components, including neural systems (Sejnowski *et. al.* 1988; Wilson and Bower 1989). A number of investigators have developed computational models of subcortical neuronal structures incorporating a substantial degree of anatomical and physiological realism. These include the cerebellum (Pellionisz and Linas 1977; Pellionisz *et. al.* 1977; Houk 1990), the hippocampus (Traub *et. al.* 1988), the pyriform cortex (Wilson and Bower 1988, 1989) and the olfactory bulb (Li 1990;Li and Hopfield 1998; Freeman 1979). These models have been useful in elucidating the mechanisms of oscillatory and elliptic activity in the hippocampus (Traub *et. al.* 1987; Traub *et. al.* 1988), and have suggested mechanisms for the distributed storage and accessing of olfactory information in pyriform cortex (Wilson and Bower 1988).

A computational model of a small path of mammalian visual cortex would be useful for a number of purposes, including understanding the dynamics of cortical activity flow, the generation of cortical oscillations by recurrent excitation, and the emergence of the receptive field properties of cortical cells from their connectivity and physiological properties (Patton, Thomas, and Wyatt 1990; Patton, Thomas, and Wyatt 1991a,b). We are currently developing such a model of visual cortex for the investigation of these questions. Among the challenges to be overcome in its creation is the generation of a network whose connectivity incorporates available knowledge of the anatomy and physiology of mammalian visual cortex. Here we describe a method of generating an appropriate pattern of connections for a model of the vertical organization of primary visual cortex. The network generated by these methods is then subjected to simulation of the input fibers and the activity of the system is followed by integrating a large system of coupled first-order nonlinear differential equations. The primary output is the activity of each neuron for hundreds of discretized time steps. The model to be described was written using Fortran code and implemented on a Stardent 2000 Graphics Mini-Supercomputer and the Cray Y-MP supercomputer.

2.0 A Model of Cortical Connectivity. The next sections of this paper discuss the various neural components and the methods by which the neural positions and connections were developed.

2.1. Overall Geometry of the Model. The slab of cortical tissue modeled here is bounded laterally by a square $500\mu m$ on each edge, extending $1500\mu m$ from the surface of the brain down to the white matter. Neurons are distributed in three dimensions through 6 cortical laminae, three of which have been further subdivided into sublaminae. The thickness assigned each lamina and sublamina approximates the data of Beaulieu and Colonnier (1983) for cat visual cortex.

The model described here is intended to represent only the features of the vertical organization of visual cortex. In future work, features of its horizontal organization, including the clustering of pyramidal neurons and their apical dendrites (Fleischhauer 1974); and long range lateral connections (Mitchson and Crick 1982; Gilbert and Wiesel 1983) will be added.

2.2. Cell and Afferent Morphologies: Stick Neurons. Each neuron in the model was given a highly schematic three dimensional axonal and dendritic arbor, consisting of line segments, so as to model a specific morphological cell type. Seven types of pyramidal cells whose differing morphologies were based on the descriptions of Lund et. al. (1979) (see Table [1]) were placed in layers 2-6 (except layer 4b). Peters and Regidor (1981) recognize eleven different morphological classes of non-pyramidal cells. Based on their descriptions and camera lucida drawings, stick arbors were devised for each of these cell classes as described in Table [1]. In all, 17 model cell types were constructed. The construction of each cell type required about 100 lines of Fortran code. Calls to DI-300 graphics subroutines (from Precision Visuals, Inc.) provided the capability to plot each stick model within the cortical slab. Plots for the various cell types used in the model are shown in Figures 1 and 2. Two major classes of cortical thalamic afferents, X and Y fibers, have their terminal arbors in visual cortex. These afferents have been included in the model, but they are modeled in a somewhat different fashion than are the intrinsic neurons listed in Table [1]. Individual thalamic afferents arborize over a broad area of cortical surface, sometimes as much as a square millimeter. In order to achieve a broad dispersal of axon terminals throughout such an arbor, we chose to abandon the convention of modeling the axonal arbors as sticks. Instead each afferent arbor is modeled as a series of cylindrical volumes extending through each lamina or sublamina in which terminal arborization has been reported.

2.3. Distribution of Neurons in Laminae. In each model cortical lamina, cells were distributed in three dimensions according to pseudorandom numbers chosen for the $x, y,$ and z coordinates. The abundance of each morphological cell class in the model was estimated using data from the experimental literature (Beaulieu and Clonnier 1983, Gabbott and Somogyi 1986, Lund *et. al.* 1983). The estimates of relative cell number used in the model for each cell class are summarized in Table [2].

2.4. Establishing Neuron Connectivity. In order to model the interactions between neurons in the model cortical network, it is necessary to specify the synaptic connectivity of the network. The location of each connection with respect to presynaptic and postsynaptic soma must be specified so that propagation times and signal strengths may be assigned. In the model, synaptic connection are assigned based on proximity between the arbor of the presynaptic neuron and the dendrite or soma of the postsynaptic cell. The model cortical slab is divided into 3000 cubical boxes, each $50\mu m$ on a side, and each labeled by a unique integer index. For each model neuron, the indices of the boxes through which the axonal arbor line segments pass are stored in an array AXON. The box indices of the boxes through which the dendritic arbor line segments pass are stored in an array DEND. The box index of each cell's soma is stored in an array SOMA. A particular neuron might have ramifications passing through hundreds of boxes. A connection is assigned whenever one cell's AXON array contains an index which is also a member of another cell's DEND or SOMA array. A connection may be thought of as representing one or several synapses. Figure 3 illustrates boxes drawn around the dendrites (thick lines) and axons (thin lines) for two neurons; the axon from the upper neuron "connects" to the dendrites from the lower neuron in the four cross-hatched boxes. It should be noted that each model neuron might be better thought of as a cluster of spatially extended boxes than as a simple assemblage of line segments, with the boxes representing the ramifying space filling fine structure of the axonal and dendritic arbors.

2.5. Connectivity Generated by the Model: The Connection Matrix. A distinctive feature of neocortex is that the connectivity of its cells is highly divergent and highly convergent (Douglas and Martin 1990; Martin 1988). A typical neuron receives synaptic input from hundreds of other cells, each of which makes only a few synaptic contacts with it. This neuron in turn projects to hundreds of other neurons, synapsing only a few times on each. About $19,600$

cortical neurons lie beneath a $500 \times 500 \mu m^2$ patch of the binocular portion of cat primary visual cortex (Beaulioeu and Colonnier 1983). This volume of cat primary visual cortex contains about 116 million synapses, amounting to an average of 6000 synaptic connections per cell (Beaulieu and Colonnier 1985).

The pattern of connections generated by the cortical anatomy model can be represented by means of a connection matrix (Figure 4). The connection matrix indicates the connectivity of each cortical cell class in the model with every other cortical cell class in the model. It consists of a rectangular array of $1,443$ boxes which together represent all possible conjunctions of the 39 possible presynaptic cell classes with the 37 possible postsynaptic cell classes (X and Y afferents synapse onto other cells but do not receive synaptic connections). The shading in the boxes indicates the value of the connection number Z , where $Z = (C_c/C_t) \times 100\%$. C_c is the number of connections which the numbers of the presynaptic cell class make with members of the postsynaptic cell class. C_t is the total number of connections occurring in the cortical slab model. White squares indicate a complete absence of connections between the cell classes in question. Darker shades indicate more connections.

The plot shows, first, that all possible connections between cell classes do not occur in the model. 592 pairs of cell classes (about 41% of the possible pairs) exhibit no connections with one another. The pattern of connections generated by the model does possess structure and specificity. Secondly, when connections do occur between cell classes the number of connections for any one cell class pair tends to be small: 771 pairs of connected cell classes each exhibit less than 0.25% of all connections. The only connections which are more abundant than this are those between the classes of pyramidal cells in the model and those between pyramidal cells and thalamic afferents. The most abundant connections in the model are represented by the input received by layer 4 and layer 6 pyramidal cells. The abundance of connections between pyramidal cell classes in the model is due to the numerical abundance of the cells themselves.

Cells in the superfical layers (1,2,3) of the model cortex receive most of their input from other cells in the superficial layers, with the layer 5 pyramidal cells supplying one the few major sources of deep input to the superficial layers. Cells in the middle(4), and deep (5,6) layers receive extensive input from throughout the depth of the cortex. Pyramidal cells in the superficial layers of the model cortex tend to receive synaptic input primarily from within these, but to send output throughout the depth of the cortex. Deep layers but pyramidal cells receive input from throughout the depth of the cortex. Layer 5 pyramidals also send output to almost every cell class of the cortex. Layer 6 pyramidal output is more restricted, with no output reaching layers 1 and 2, and only basket cells receiving layer 4 pyramidal input in layer 3a.

3.0 Modeling Cortical Physiology. In the upcoming sections we discuss the development of the physiological dynamics and its subsequent relationship to the cortical connections developed in the previous discussion.

3.1. Cell Physiology. A great deal of experimentally derived physiological data was used to simulate the properties of cortical neurons in this model. All membrane potentials were scaled to the range [0.0-1.0].These scaled voltages are referred to as activities, with the correspondence between membrane potential M and activity V given by:

$$V = \frac{(M + 88mV)}{133mV} \tag{1}$$

The lower bound, $V = 0.0$, corresponds to $M = -88mV$, which is taken to be the lowest membrane potential reached by a fast spiking cell following development of the action potential. The upper bound, $V = 1.0$, corresponds to a membrane potential of $M = +45mV$, the average value at peak height of an action potential spike (McMormick *et.al.*, 1985).

In the dynamical equations, all activities were initialized at rest. An average resting membrane potential of $M = -75mV$ was used (Connors *et. al.* 1982), which corresponds to a rest activity of $V_0 = 0.1$. The firing threshold used is $M = -50mV \pm 9mV$ (Connors *et. al.*, 1082), which

corresponds to the activity range $V = 0.286 \pm 0.067$. The threshold activity for each neuron was chosen at random from within this interval.

Deviations from the rest activity arise once signals start to arrive externally from the thalamus and intracortically from other spiking neurons. Excitation or inhibition, however, does not reach a postsynaptic neuron instantaneously. The model incorporates a delay in signal arrival that is made up of both the time taken to travel along the presynaptic axon, as well as a synaptic delay. A minimum synaptic delay of 0.5msec was used, based on findings for neuro-muscular junctions (Aidley 1978;Shepherd 1979).

A synaptic delay in the range from 0.5 to 1.0 msec was randomly assigned for each connection. The speed of signal conduction along the slow X fibers from the thalamus ranges from 18-25msec and along the fast $Y-$ fibers from 30-40msec (Stone and Dreher 1982). Each X and Y fiber in the model was randomly assigned a conduction velocity from within these ranges. The axons of intracortical neurons in the model were assigned velocities randomly selected between 0.3 and 0.6msec (Swadloe *et. al.,* 1978; Swadlow, 1974). Signal propagation along a dendrite was taken to be passive electronic spread and therefore to be practically instantaneous.

3.2. Equations for Cell Activities. The equations governing the time dependence of the activity for each neuron depend upon whether the activity, $V_i(t)$, is below or above the firing threshold, θ_i , for the neuron. These two dynamic regimes have an important functional significance in the network. In the suprathreshold regime, the neuron sends signals over its axon to other cells, but does not receive signals. By contrast, in the subthreshold regime, the neuron receives signals from other neurons, but cannot transmit signals. Within a short time interval, signals arrive from a relatively small number of firing neurons. These two regimes are considered below.

a. **Subthreshold regime,** $V_i(t) < \theta_i$

In this case, all of the incoming signals to neuron i are summed together in the cell body. The rate of change in the activity is then given by the coupled system of 1st order nonlinear differential-delay equations:

$$\frac{dV_i}{dt} = -\frac{(V_i(t) - V_0)}{\tau} + \sum\nolimits_i^{(+)}(t) - V_i(t)\sum\nolimits_i^{(-)}(t) \tag{2}$$

where

- $-\frac{(V_i(t)-V_0)}{\tau}$ is the relation term which acts to return V_i to the rest activity, V_0 . The relaxation rate is governed by τ , the relaxation time. In this study, the value $\tau = 7$ msec was used (Connors *et. al.,* 1982; Tetusa *et. al.,* 1981).
- $\sum_i^{(+)}(t)$ is the total excitatory input, including both thalamic and intracortical contributions.
- $\sum_i^{(-)}(t)$ in the total inhibitory input, from cortical inhibitory presynaptic neurons.
- $-V_i(t)$ gates the inhibitory input and prevents $V_i(t)$ from dropping below the value 0.

The two input terms, $\sum_i^{(+)}(t)$ and $\sum_i^{(-)}(t)$, are given by summing over all presynaptic neurons to which neuron i is connected. Connections neuron i receives from neuron j are divided into two categories; somatic and dendritic connections. Currently, this distinction is made solely for the purposes of assigning different propagation times for the two cases. No distinction in terms of connection strength is made. The summation is as follows:

$$\sum\nolimits_i^{(+)}(t) = \sum_j k_{ij}(n_{ij}^{(s)}S(t - \Delta_{ij}^{(s)}) + [n_{ij}^{(d)}S(t - \Delta_{ij}^{(d)})] \tag{3}$$

where

- $n_{ij}^{(s)}, n_{ij}^{(d)}$ are the number of somatic or dendritic connections between neurons i and j .
- k_{ij} is the connection strength (synaptic efficacy) between neurons i and j . In the present study, the connection strength is held fixed at a value of 0.05 for all connections ij . Continuous activation of such a synapse for 1 msec would be sufficient to raise a typical model neuron to firing threshold.

- $S(t - \Delta_{ij})$ is the signal function which allows for delay in the signal leaving neuron j and arriving at neuron i. The signal function is 1 if neuron i spiked at the earlier time $(t - \Delta_{ij})$, and is zero otherwise.
- Δ_{ij} is the average signal delay time (for either somatic or dendritic connections), which includes the average propagation time for axonal condition between the soma of neuron j and the connection, plus a random synaptic delay time between 0.5 and 1msec. In general, neuron j makes multiple connections onto neuron i and the delay time is calculated by averaging over the delay times for the individual paths.

The dynamical equations, together with the initial conditions and specification of the thalamic input signals, define an initial value problem. In this study, the thalamic input fibers are pulsed between t_0 and $t_0 + \Delta$, with $t_0 = 0.2$msec and $\Delta = 0.1$msec. The system of differential equations was integrated with the forward Euler algorithm, with a time step of 0.03msec or 0.05msec.

b. **Suprathreshold regime**, $V_i(t) > \theta_i$

When the activity for neuron i exceeds the threshold value, the differential equation for this neuron is not integrated during the firing sequence. Instead, the activity is computed from an analytic spike function, $G_i(t - t_s)$, where t_s is the time at which this neuron reached threshold. $G_i(\tau)$ (where $\tau = t - t_s$)) has different values, depending upon the type of neuron that is firing and the firing state that it is in. Three types of action potential patterns are described elsewhere (Patton *et. al.* 1991).

After firing, each cell, based on the experimental findings of Anderson et. al. (1978), and Waxman and Swadlow (1977), was given an absolute refractory period of 1msec. The cell remained hyperpolarized during this period. After the end of the absolute refractory period, the cell's membrane potential was again governed by the subthreshold regime equations, and, in the absence of other inputs, recovered from hyperpolarization in about 9 msec.

4.0 Dynamical Behavior of the Model.

4.1. Simulation Conditions. The results reported here are the product of a series of simulation runs in which we sought to characterize the response of the cortical slab to the delivery of a brief stimulus to the thalamic afferents. The stimulus was a 0.1msec long pulse delivered simultaneously to each thalamic afferent at the base of the cortical slab. The stimulus paradigm was intended to model the delivery of an electrical stimulus to the fibers leading from the thalamus to the cortex, as occurs in experiments aimed at determining neuronal response latencies. Except where stated otherwise, the model slab contained 5% of the true number of neurons contained in such a slab of cat visual cortex, or about 980 cells (Beaulieu and Colonnier 1985). All synaptic weights k_{ij} were set equal to the value 0.05. At this value, continual activation of one connection for 1msec is sufficient to raise a cell to firing threshold. The response of the slab to the stimulus was computed using integration time steps of 0.05msec.

4.2. Behavior of the Model Cortical Slab Under Stimulation. The program DYNAMIC competes the activity of each neuron in the model slab as a function of time. Activities of individual selected neurons can be displayed (Patton *et. al.* 1990). Another way of representing the behavior of the model cortical slab in response to thalamic afferent stimulation is in terms of average activity levels rather than numbers of firing cells. In Figure 5, average activity is shown as a function of cortical depth and time for a period of 4.9msec or 98 integration time steps. The vertical extent of the $1500\mu m$ slab was divided into 30 $50\mu m$ thick levels. The mean activity in each level was computed by summing the activity of the neurons in the level and dividing by the number of such neurons. The graph shows that activity in the model begins earliest in the cortical layers where thalamic afferents arborize; 4a, 4b, and 6, with a mean activity above 0.15 being reached about 0.6msec after stimulation. Activity then spreads to layer 5 and the superficial layers, with the 0.15 mean activity level being reached about 1.5msec after stimulation in layers 5 and 2+3a, and about 2.7msec after stimulation in layer 1. The time delay for spread of activity in the cortical slab has been compared to experimental data (Patton *et. al.* 1991a); the agreement is quite good.

4.3. Radiosity Rendering of Dynamic Data. An excellent method to visualize neurobiological data has recently been developed by Driver and Buckalew (1991), Witten(1990). Using radiosity rendering methods, cell bodies in the cortical slab are given a radiative intensity that is keyed to the neuron activity. When the neuron activity goes above threshold, the neuron changes color from blue to red and then yellow to white as the peak firing activity is reached. In addition to this change of color, the radiative intensity of the small polyhedron representing the cell body increases rapidly with cell activity, so that the neuron appears to emit increasingly intense white light as the peak activity is reached.

Due to our inability to include color pictures, we are unable to illustrate an actual picture of the simulation. Videotape is available from the first author. The figure is a rectangular slab, divided into six layers. The neurons are distributed throughout the slab and illuminate as a function of their activity at a given frame time point. The color of the neuron is proportional to the activity. When an activity exceeds a predefined voltage, the neuron produces a burst of light and, using the methods of radiosity, is seen to glow for the period of time it exceeds the firing threshold. Each frame also shows a color - voltage reference bar so that the viewer may observe the level of activity as a function of color. Finally, time step and level activity are also displayed. To the right of the cortical slab is shown the color scale which ranges from blue at low activity to white at peak activity. To the far left of the slab is a time bar, which increases vertically as time increases from the start of the simulation sequence. In addition, to the immediate left of the slab is plotted the average neuron activity in each of 30 $50\mu m$ thick horizontal slices through the slab. In this simulation, the cortical slab contains about 600 neurons, some of which are firing. As they do so, white light is emitted and illuminates the surrounding neurons. Frames of the sequence were rendered at the rate of about 3 frames/minute. Each frame was stored on videodisk, which permitted the entire sequence of portions thereof to be viewed interatively.

5.0 Mapping From Simulation to Symphony. In viewing the videotape of the simulation of this model, we realized that a musical track would be necessary. First, the track was viewed as being merely background music. Various tracks were used and most of the viewers considered Vivaldi's *Concerto in B minor for Four Violins*, Op.3 No.10 an appropriate choice. An equally interesting alternative was to use Wendy Carlos's/Benjamin Folman's *Switched-On Bach* Brandenberg Concerto No.3. The combination of listening to these two musical tracks and our need to extend the information content of the soundless video lead to the concept of letting the simulation generate its own electronic audio track. The ear is an extremely useful pattern analysis system. Conceptually, if the simulation were biologically accurate, it might be that different organic brain disorders, when modeled might be not only visually different, but different in their musical signature. In addition, the combination of visual information, coupled to an audio actualization, might provide information that neither would provide separately.

Previously, we discussed the concept of visualization descriptors. We pointed out that there were two obvious descriptor types; *static* and *dynamic*. For example, in our neural simulation, we might consider the set $\mathcal{D}_S = \{\mathcal{P}_n, \mathcal{T}_n, \mathcal{L}_n\}$ where $\mathcal{P}_n = \{(x, y)$ position of the nth neuron$\}$, $\mathcal{T}_n = \{$ neuron type of nth neuron $\}$, $\mathcal{L}_n = \{$level or layer position of the nth neuron$\}$. A set $\mathcal{D}_D$ of dynamic descriptors might be the voltage of the nth neuron and the radiosity or glow level of the nth neuron.

In order to build a BioSymphonic visualization environment, we clearly need the concept of a `DESCRIPTOR` and an associated `DESCRIPTOR⊨TYPE`. These concepts are tied down using an adjectival predecessor for the `DESCRIPTOR` . For example, `MODEL⊨DESCRIPTOR`. Hence, within the context of the cortical model, we might define `MODEL⊨DESCRIPTOR NEURON-TYPE`. Associated with each `DESCRIPTOR` is the `DESCRIPTOR⊨TYPE`. For `NEURON-TYPE`, we might consider assigning it a `STATIC` type as it will not change during the course of the visualization process. Hence, we would make use of the command `MODEL⊨DESCRIPTOR⊨TYPE NEURON-TYPE : STATIC`. The introduction of labels and label types automatically implies that the system must allow editing of both label and label types. In addition, we have built into the system a type checking between sonic labels and model descriptor lables. For example, we would not assign voltage to MIDI note number. This is a type mismatch as we would normally assign the type as follows `SONIC⊨DESCRIPTOR MIDI-NOTE-NUMBER`

and SONIC⊨DESCRIPTOR⊨TYPE: STATIC.

To assign a particular model descriptor to a particular visualization descriptor, we use the LINK command, followed by the two descriptors to be linked. For example, LINK MIDI-NOTE-NUMBER TO NEURON-TYPE. If there are any predefined links or type mismatches, the system will warn the user in an X-windows ERROR MESSAGE WINDOW. Other windows include LINK STATUS and EDIT OBJECT where OBJECT may be any DESCRIPTOR or DESCRIPTOR⊨TYPE. The details of the system are available, in preprint form, in another paper.

The system is developed for a UNIX, X-windows environment. It currently runs over a network of SUN SPARC Station IPC's, 1+'s. These SUNS are connected to a CRAY YMP8/864 (for simulation and numerical computation), and a CONVEX C220 (for database access).

For our initial simulations, we chose to map the cortical levels onto octaves, the neuron type onto a twelve tone scale in a given octave, and the volume onto the firing voltage. This leads to a somewhat confusing musical scenario. We are currently in the process of developing a more complex mapping that will allow for additional musical features such as attack and delay to be mapped onto the simulation.

6.0 Acknowledgments. For access to the Cray Y-MP8/864 supercomputer, we thank the University of Texas System Center for High Performance Computing. In addition, we thank Cray Research Inc. for two years of grant support.

7.0 References.

[1] AIDLEY, D.J., **The Physiology of Excitable Cells** *(Cambridge University Press, Cambridge, 1978).*

[2] AIKIN, J., **Midi Sequencing For Musicians** (GPI Publications, Cupertino, CA 1989).

[3] ANDERSON, P.H., SILFVENIUS, S.H., SUNDBERG, O.SVEEN AND H. WINGSTRON, *Functional characteristics of unmyelinated fibers in the hippocampal cortex*, Brain Res., **144:**11-18, 1978.

[4] BALZANO, G.J., *The group-theoretic description of 12-fold and microtonal pitch systems*, Computer Music Journal, 4#4 (1980) 66-84.

[5] BEAULIEU, C. AND COLONNIER, M., *The number of neurons in the different laminae of the binocular and monocular regions of area 17 in the cat*, J. Comp. Neurol. **217:**337-344, 1983.

[6] BEAULIEU, C. AND COLONNIER, M., *A laminar analysis of round asymmetrical and flat - asymmetrical synapses on spines, dendritic trunks and cell bodies in area 17 of the cat*, J. Comp. Neurol, **231:** 180-189, 1985.

[7] BEGAULT, D.R., AND WENZEL, E.M., *Techniques and applications for binaural sound manipulation in human - machine interfaces*, NASA Technical Memorandum 102826, August 1990.

[8] BEGAULT, D.R., AND WENZEL, E.M., *Technical aspects of a demonstration tape for three - dimensional sound displays*, NASA Technical Memorandum 102826, October 1990.

[9] BUXTON, B., *Using our ears: An introduction to the use of nonspeech audio cues*, **SPIE vol. 1259 Extracting Meaning From Complex Data: Processing, Display, Interaction**, pp. 154-162, 1990.

[10] CONNORS, B.W., GUTNICK, M.J. AND PRINCE, D.A., *Electrophysiological properties of neocortical neurons in vitro*, J. Neurophysiol, **48:**1302-1335, 1982.

[11] DEFURIA, S., AND SCACCIAFERRO, J., **The Midi Book** (HAL Leonard Books, Milwaukee, 1988).

[12] DOUGLAS, R.J. AND MARTIN, K.A.C., **Neocortex In The Synaptic Organization of the Brain**, G.M. Shepherd(ed.) (Oxford University Press Inc.), pp.389-438, 1990.

[13] DRIVER, J. AND BUCKALEW, C., *Radiative tetrahedral lattices*, **Proc. SPIE vol. 1459 Symposium On Extracting Meaning From Complex Data: Processing, Display, Interaction, 1991** in press.

[14] EVANS, B., *Correlating sonic and graphic materials in scientific visualization*, **SPIE vol. 1259 Extracting Meaning From Complex Data: Processing, Display, Interaction**, pp. 154-162, 1990.

[15] FLEISCHHAUER, K., *On the different patterns of dendritic bundling in the cerebral cortex of the cat*, Z. Anat. Entwickl.Gesch., **143:**115-126, 1974.

[16] FREEMAN, W.J., *Non-linear dynamics of paleocortex manifested in olfactory EEG*, Bio. Cyber, **35:** 221-234, 1979a.

[17] FRYSINGER, S.P., *Applied research in auditory data representation*, **SPIE vol. 1259 Extracting Meaning From Complex Data: Processing, Display, Interaction**, pp. 154-162, 1990.

[18] FUCHS, H., LEVOY, M. AND PIZER, S.M., *Interactive visualization of 3D medical data*, IEEE Computer, August 1989, pp. 46-51.

[19] GILBER, C.D. AND WIESEL, T., *Clustered intrinsic connections in cat visual cortex*, J. Neurosci., **3:**1116-1133, 1983.

[20] HOUK, J.C., *Modeling the Cerebellum in Neural Computation*, A.I.Selerson (ed.), Society for Neuroscience, Washington D.C., 1990.

[21] LI, Z., *A model of olfactory adaptation and sensitivity enhancement in the olfactory bulb*, Bio. Cyber **62:**3349-361, 1990.

[22] LI, Z. AND HOPFIELD, J.J., *Modeling of the olfactory bulb and its neural oscillatory processing*, Bio. Cyber **61:** 379-392.

[23] LORRAIN, D., *A panoply of stochastic 'cannons',*, preprint.

[24] LUND, J.S., HENRY, G.H., MACQUEEN, C.L. AND HARVEY, A.R., *Anatomical Organization of the Primary Visual Cortex (Area 17) of the cat: A comparison with Area 17 of the macaque monkey*, J. Comp. Neuro, **184:**599-617, 1979.

[25] LUNNEY, D. AND MORRISON, R.C., *Auditory presentation of experimental data*, **SPIE vol. 1259 Extracting Meaning From Complex Data: Processing, Display, Interaction**, pp. 154-162,1990.

[26] MARTIN, K.A.C., *From single cells to simple circuits in the cereberal cortex*, Quart. J. Exp. Physiol., **73:**637-702, 1988.

[27] MCCORMICK, D.A.,CONNORS, B.W., LIGHTHALL, J.W. AND PRINCE, D.A., *Comparative electrophysiology of pyramidal and sparsely spiny stellate neurons of the neocortex*, J. Neurophysiol., **54:** 782-806, 1985.

[28] MITCHISON, G. AND CRICK, F., *Long axons within the striate cortex: their distribution, orientation, and patterns of connection*, Proc. Nat. Acad. Sci. USA,**79:**3661-3665, 1982.

[29] PATTON, P., THOMAS, E. AND WYATT, R, *A computational model of the vertical anatomical organization of primary visual cortex*, Bio. Cyber, (in preparation), 1991a.

[30] PATTON, P., THOMAS, E. AND WYATT, R, *Computational model for space-time signal propagation in the visual cortex: II Dynamics of activity flow*, Bio. Cyber, (in preparation), 1991b

[31] PELLIONISZ, A. AND LLINAS, R., *A computer model of cerebellar Purkije cells*, Neurosci.,**2:**37-48, 1977.

[32] PELLIONISZ, A., LLINAS, R. AND PERKEL, D.H., *A computer model of the cerebellar cortex of the frog*, Neurosci., **2:**19-35,1977.

[33] PETERS, A. AND REGIDOR, J., *A reassessment of the forms of nonpyramidal neurons in area 17 of cat visual cortex*, J. Comp. Neuro., **203:** 685-716, 1981.

[34] RABENHORST, D.A., FARRELL, E.J., JAMESON, D.H., LINTON, T.D., AND MANDELMAN, J.A., *Complementary visualization and sonification of multidimensional data*, **SPIE vol. 1259 Extracting Meaning From Complex Data: Processing, Display, Interaction**, pp. 154-162,1990.

[35] RASCH, R.A., *Timing and synchronization in ensemble performance*, (in) **Generative Processes In Music** (ed.) J.A. Sloboda (Clarendon Press, Oxford, 1988).

[36] SEJNOWSKI, T., KOCH, C. AND CHURCHLAND, P.S., *Computational neuroscience*, Science, **141:** 1299-1306, 1988.

[37] SHEPHERD, G.M.(ED.), **The Synaptic Organization of the Brain**, (Oxford University Press, New York, 1979).

[38] STONE, J. AND DREHER, B., *Projections of x and y cells of the cat's lateral geniculate nucleus to areas 17 and 18 of visual cortex*, J. Neurophys, **36**:551-567.

[39] SUNDBERG, J., *Computer synthesis of music performance,*, (in) **Generative Processes In Music** (ed.) J.A. Sloboda (Clarendon Press, Oxford, 1988).

[40] SWADLOW, H.A., *Systemic variations in the conduction velocity of slowly conducting axons in rabbit corpus callosum*, Exp. Neurol,**43:** 445-451, 1974.

[41] SWADLOW, H.A., WEYAND T.G. AND WAXMAN, S.G., *The cells of origin of the corpus callosum in rabbit visual cortex*, Brian Res., **156**:129-134, 1978.

[42] TRAUB, R.D, MILES, R. AND WONG, R.K.S., *Large Scale Simulations of the Hippocampus*, ISSS Eng. in Med. and Biol., **7**:31-51, 1988.

[43] WAXMAN, S.G. AND SWADLOW, H.A., *The condition properties of axons in central white matter*, Prog. in Neurobiol., **8**:297-324, 1988.

[44] WENZEL, E.M., FOSTER, S.H., WIGHTMAN, F.L. AND KISTLER, D.J., *Realtime digital synthesis of localized auditory cues over headphones*, **Proc. 1989 IEEE ASSP Workshop On Applications Of Signal Processing To Audio and Acoustics.**

[45] WENZEL, E.M., STONE, P.K., FISHER, S.S., AND FOSTER, S.H., *A system for three - dimensional acoustic visualization in a virtual environment workstation*, **Visualization '90 Conference Proceedings**, San Francisco, CA 1990.

[46] WILKINSON, S.R., **Tuning In: Microtonality In Electronic Music** (HAL Leonard Books, Milwaukee, 1988).

[47] WILSON, M.A. AND BOWER, J.M., *A computer simulation of olfactory cortex with functional implications for storage and retrieval of olfactory information*, **Neural information processing systems** D.Z.Anderson (ed.), American Institute of Physics, New York, 1988.

[48] WILSON, M.A. AND BOWER, J.M., **The Simulation of Large-Scale Neural Networks In Methods in Neuronal Modeling**, C. Koch and I. Segev (eds.), (MIT Press, Cambridge, Mass.,1989).

[49] WINSOR, P., AND DELISA, G., **Computer Music In C** (Windcrest Books, Blue Ridge Summit, PA 1991).

[50] WITTEN, M., *Light bursts help visualize firing of neurons*, Pixel **1#4** (1990) 10.

Matthew Witten, a native of Boston, obtained his AB (Mathematics and Physics) at Boston University. After receiving an MS in Mathematical Physics, and an MS and Ph.D. degree in Mathematical Biology from SUNY at Buffalo, Center For Theoretical Biology, he spent postdoctoral research time at the Medical University of South Carolina and the University of Southern California. He is currently Director of Applications Research and Development and Associate Director of the University of Texas System Center For High Performance Computing. In addition, he holds an Associate Professor of Physiology appointment at the University of Texas Health Science Center, San Antonio.

Robert E. Wyatt, a native of Chicago, obtained his B.S. at the Illinois Institute of Technology. After receiving and M.A. and a Ph.D. in Chemistry from the Johns Hopkins University, he completed two years of postdoctoral research in England and at Harvard University. He is currently the W.T. Doherty Professor of Chemistry and Physics at the University of Texas at Austin, and Director of the Institute for Theoretical Chemistry

TABLE 1

MORPHOLOGIES AND LAMINAR DISTRIBUTION OF CELL CLASSES AS USED IN MODEL[a]

CELL TYPE (E-excitatory) (I-inhibatory)	SOMA IN LAYER	Schematic morphologies as used in model — DENDRITES	AXON
horizontal	1	clustered in the direction of the axon, or the opposite direction, length 125μ	horizontal axon, from one side of soma; length 100-150μ
basket(I)	2,3, 4,5	6 dendritic branches radiating from soma; length 75-225μ	axon rises vertically from soma, length 75μ; then 6 branches, lengths from 75 to 115μ, located 70-100° off the vertical
basket cell of layer 6	6	12 dendrites, length 150μ, radiate from soma	axon up to layer 4, length 500-750μ, then 15 branches, length 100μ, in layer 4
chandelier(I)	2,3, 4	6 dendritic branches radiating from soma; length 100-200μ	axon drops 200μ from soma; there are 6 horizontal branches, length 50-150μ
sparsely spinous bitufted (double bouquet)(I)	2,3	6 dendritic branches radiating from soma; length 100-200μ	axon drops 200μ from soma; there are 10 horizontal branches, length 50-100μ
spherical multipolar(I)	2,3,4, 5,6	20 dendritic branches spherically symmetric about soma; length 50-75μ	10 axonal branches spherically symmetric about soma; length 75-100μ
horizontal (I) cell of layer 6	6	main dendrites, length 150μ, from opposite sides of soma; at end of each branch, have one angular branch, length 125μ	axon descends to white matter
bipolar(I)	4,5	3 dendritic branches, length 150-250μ, angle <20° or >160° from vertical	axon drops 200μ from soma, then 2 horizontal branches; length 25-50μ
large spinous stellates(E)	4a	10 dendritic branches, length 125μ	axon drops 40μ from soma, 50-100μ horizontal, then vertically up to layers 2 & 3
small spinous stellates(E)	4b	10 dendritic branches, length 125μ	axon drops 40μ from soma,50-100μ horizontal, then vertically up to layer 4a
pyramidal(E)	2 & 3a	apical dendrite up to layers 1 & 2, then 6 branches, length 120μ; 4 basal dendrites, length 100-150μ	axon down to white matter (to the 18, 19, and Clare-Bishop areas); axonal branching in layer 5b (branch length 50-100μ); axon collateral starts 100μ below soma, horizontal branch length 50μ, then vertical (length 100-200μ) projection up to layers 2&3a
pyramidal(E) (large)	3b	apical dendrite up to layers 2 & 3a, then 6 branches, length 120 μ; 4 basal dendrites, length 100-150μ	axon down to white matter (to the 18, 19, and Clare-Bishop areas); axon down to layer 5a, then branching in 5a, length 50-200μ; axon collateral starts 100μ below soma, 50μ horizontal, 100-200μ to layer 2&3a
star pyramidal(E)	4a	apical dendrite up to layers 1 & 2; 4 basal dendrites, length 100-150μ	axon drops down to white matter; collateral starts 100μ below soma, runs 50μ horizontal, then up to layers 2&3a
pyramidal(E) (small to medium)	5a	apical dendrite up to layers 1 & 2, 6 branches in 1 & 2, length 120μ; 4 basal dendrites, length 100-150μ	no axonal projection to white matter; axon up to layers 2&3a; collateral drops 50μ from soma, then 50μ horizontal, with vertical projection up to layers 2&3a; in layers 2&3a branches up to 150μ long
pyramidal(E) (large)	5b	apical dendrite up to layers 2 & 3, then 6 branches, length 120μ; 4 basal dendrites, length100-150μ	axon down to white matter; collateral drops 50μ from soma, then 50μ horizontal, then vertical up to layers 2 & 3a; in 2 & 3a branches 150μ long
pyramidal(E)	6	apical dendrite up to layers 2 & 3, then 6 branches, length 120μ; 4 basal dendrites, length 100-150μ	axon to white matter (LGN); collateral to layer 4 starts 50μ below soma, then 50μ vertically up to layer 4
pyramidal(E) (nonprojecting)	6	apical dendrite up to layer 3, basal dendrites, length 50-150μ, 4 basal dendrites, length 50-150μ	axon in layer 6

[a]Morphologies for pyramidal neurons based on Lund et. al. (1979). Morphologies and laminar distributions of non-pyramidal cell classes based on Peters and Regidor (1981).

TABLE 2

CELL DISTRIBUTION USED IN MODEL

layer	cell type[a] and %						layer sum
1	h1						
	1.7						1.7%
2	p2	sm2	b2	c2	db2		
	7.2	1.0	0.4	0.1	1.0		9.7%
3a	p3a	sm3a	b3a	c3a	db3a		
	8.5	1.1	0.4	0.2	1.0		11.2%
3b	p3b	sm3b	b3b	c3b	db3b		
	9.4	1.4	0.5	0.2	1.4		12.9%
4a	p4a	s4a	sm4a	b4a	c4a	bp4a	
	11.8	2.0	1.3	0.6	0.2	1.4	17.3%
4b	s4b	sm4b	b4b	c4b	bp4b		
	14.1	1.6	0.1	0.2	0.6		17.6%
5	p5	b5	bp5				
	6.3	0.3	0.9				7.5%
6a	p6a	pn6a	em6a	sm6a			
	13.7	3.9	0.9	0.8			19.3%
6b	p6b	pn6b	h6b				
	2.1	0.4	0.3				2.8%

excitatory cells = 79.4% inhibitory cells = 20.6% Total = 100%

[a]Notation for cell types (based on the terminology of Peters and Regidor (1981)).

b	basket
bp	bipolar
c	chandelier
db	double bouquet (sparsely spinous bitufted)
em	elogate multipolar (basket cells of layer 6)
h	horizontal
p	pyramidal
pn	pyramidal in layer 6, does not project to dLGN
s	spinous stellate
sm	spherical multipolar (small multipolar and sparsely spinous stellate cells, sparsely spinous cells of layer 6)

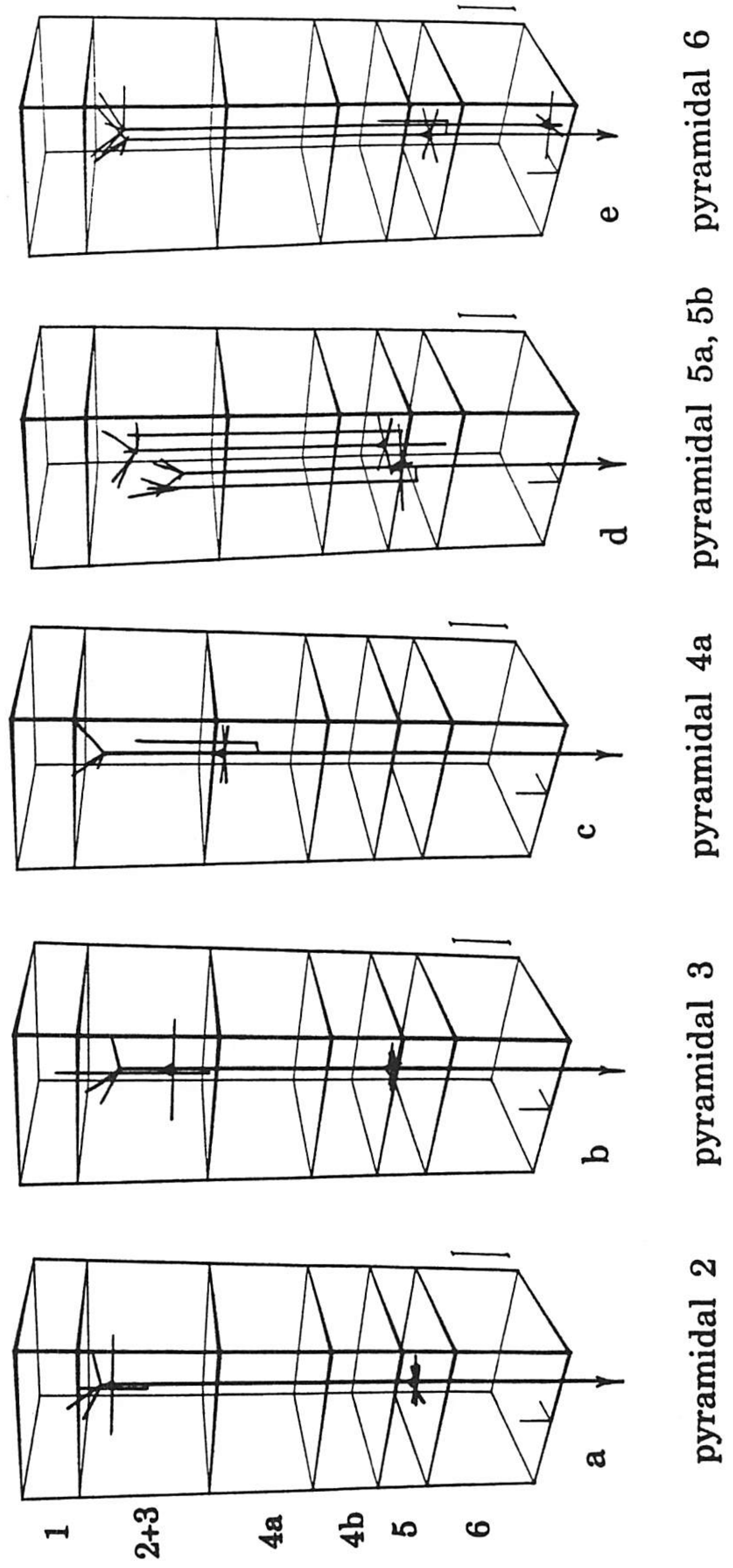
Figure[1] Schematic morphologies of cortical pyramidal cells in the model for layers 2-6. The construction of the line segment arbors is based upon data presented in Lund (1979), The neurons are shown embedded within a rectangular cortical slab which extends through the entire depth of the cortex and is $500\mu m$ on a side. Laminar boundaries are indicated. The location of each lamina is shown in 1a. A scale marker to the right of each slab indicates a distance of $200\mu m$.

Figure[2] Schematic morphologies of cortical non-pyramidal cells as used in model. The construction of the line segment arbors is based upon the descriptions of Peters and Regidor (1981). The model neurons are shown embedded in a cortical slab as in Figure 1. The cell types plotted, and the layers in which they occur, is as indicated below each slab.

Figure[3] The shaded boxes illustrate how *box connections* between the axon (thin lines) from the upper neuron are established with the dendrites (thick lines) from the lower neuron.

Figure[4] The connection matrix shows the number of synaptic connections occurring between each possible pairing of cell classes in the model. Presynaptic cell classes are indicated on the left, the postsynaptic cell classes at the top. The gray level in each box indicates the value of the connection number Z (as defined in the text) for the given pair of cell classes. The gray scale at the right indicates the range of Z values associated with each gray level. Only pairs of cell classes with a conne tion number of 0 are indicated by white squares. Pairs having connection numbers > 2.50 are indicated by black squares. Intermediate gray levels represent bins having the indicated upper bound. The number of pairs of cell classes associated with each connection number gray level is indicated.

Figure[5] Contour map showing average activity in the cortical slab following stimulation for 0.1msec at the time indicated by 'stim'. The vertical axis shows cortical laminae 1-6, while time (in msec) following stimulation is shown on the horizontal axis. The contours (0.15, 0.25, 0.35, 0.45, and 0.55) show the spread of activity through the laminae.



1
2+3
4a
4b
5
6
a
pyramidal 2
b
pyramidal 3
c
pyramidal 4a
d
pyramidal 5a, 5b
e
pyramidal 6

Fig. 1

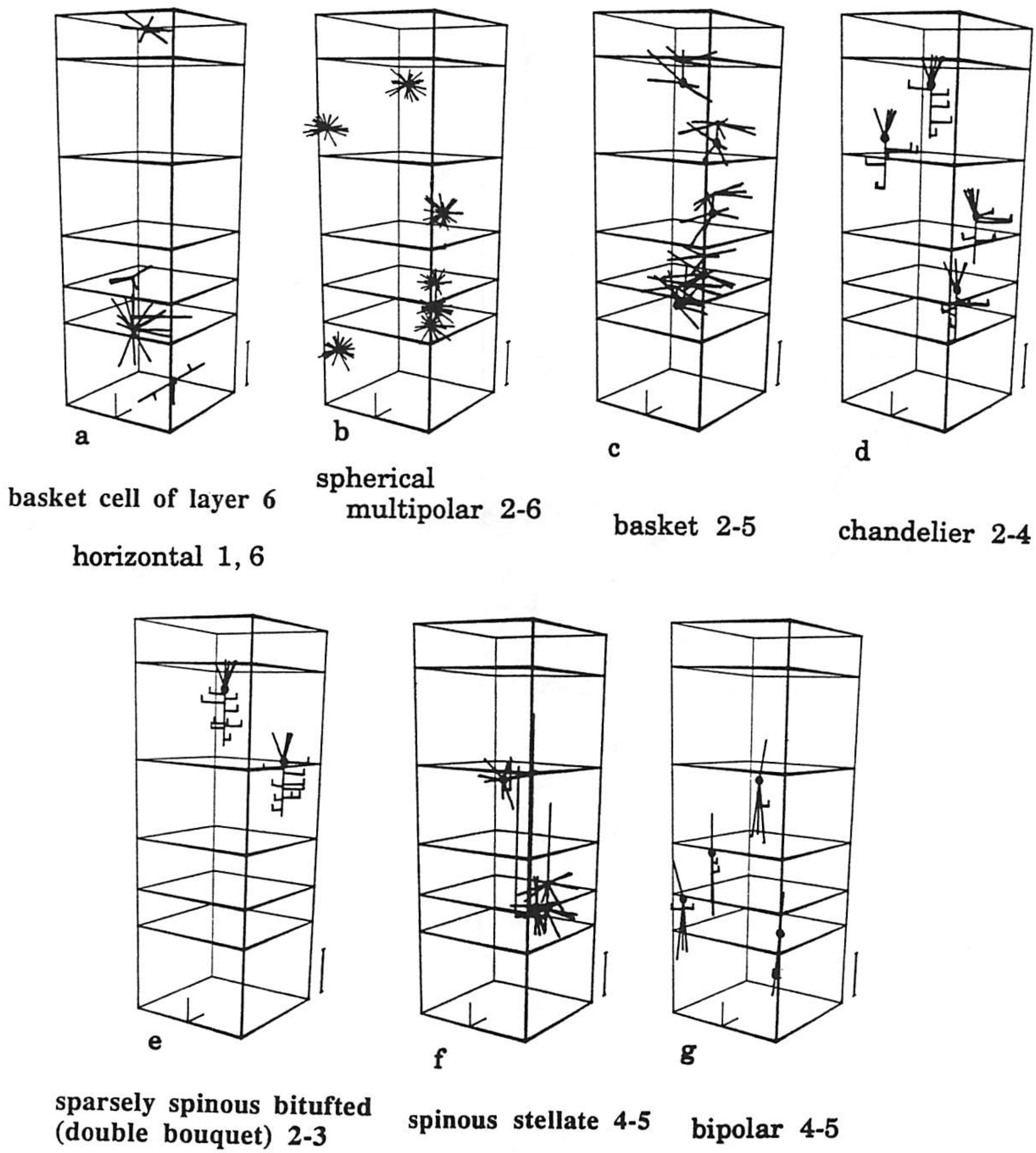

a
basket cell of layer 6
horizontal 1, 6
b
spherical
multipolar 2-6
c
basket 2-5
d
chandelier 2-4
e
sparsely spinous bitufted
(double bouquet) 2-3
f
spinous stellate 4-5
g
bipolar 4-5

Fig. 2

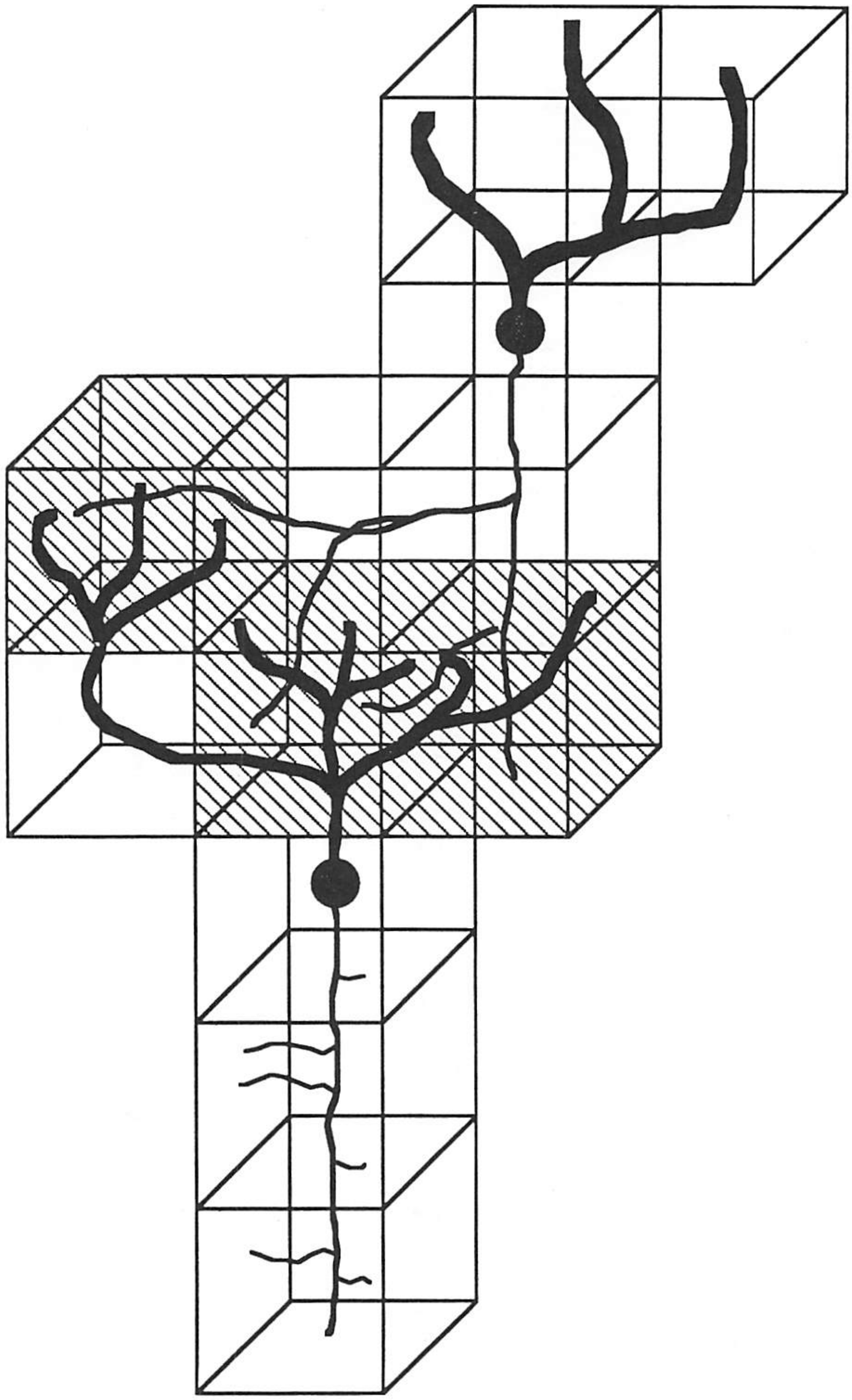
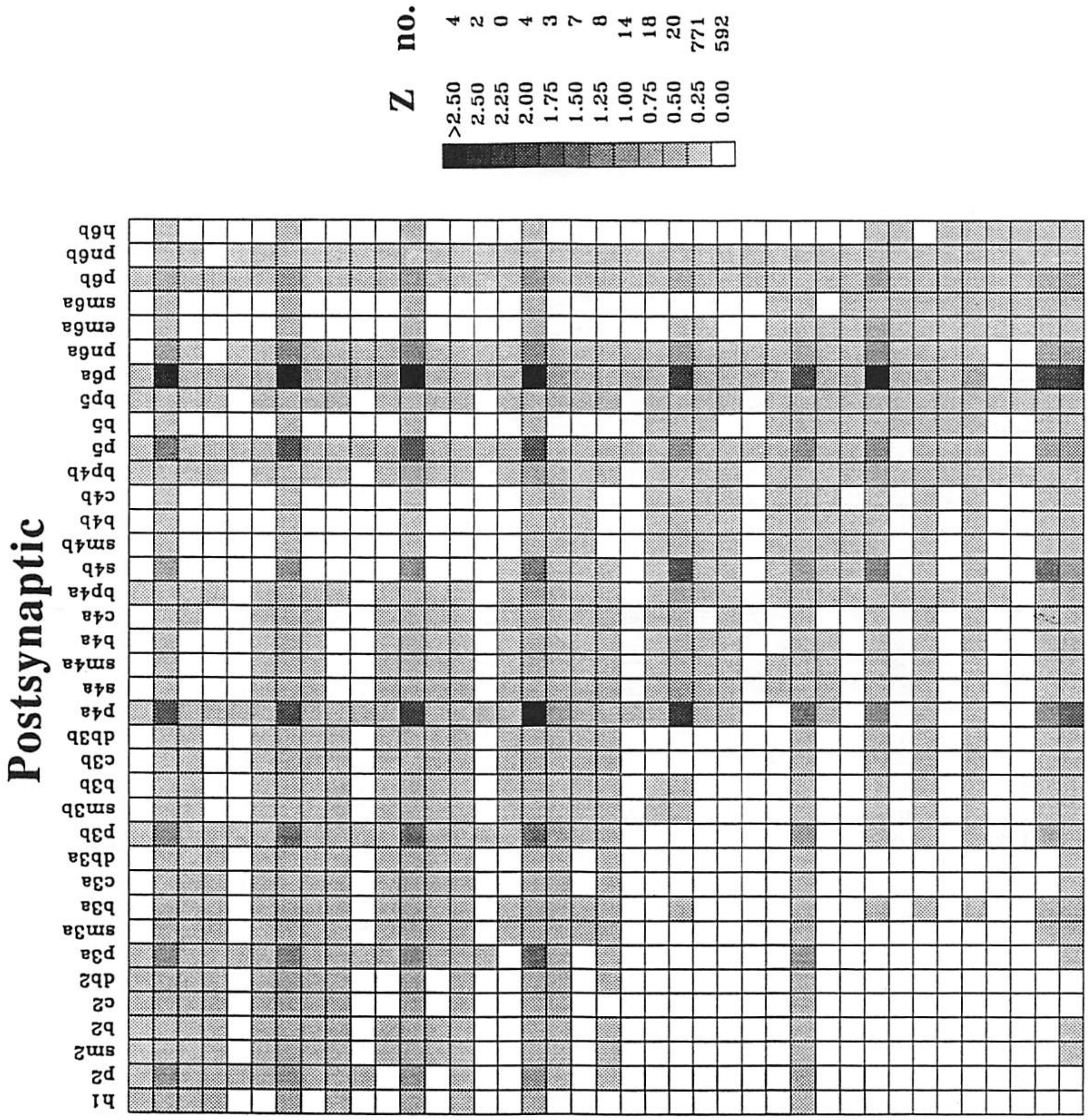


Fig. 3



Postsynaptic
Presynaptic
Z no.
>2.50 4
2.50 2
2.25 0
2.00 4
1.75 3
1.50 7
1.25 8
1.00 14
0.75 18
0.50 20
0.25 771
0.00 592

Fig. 4

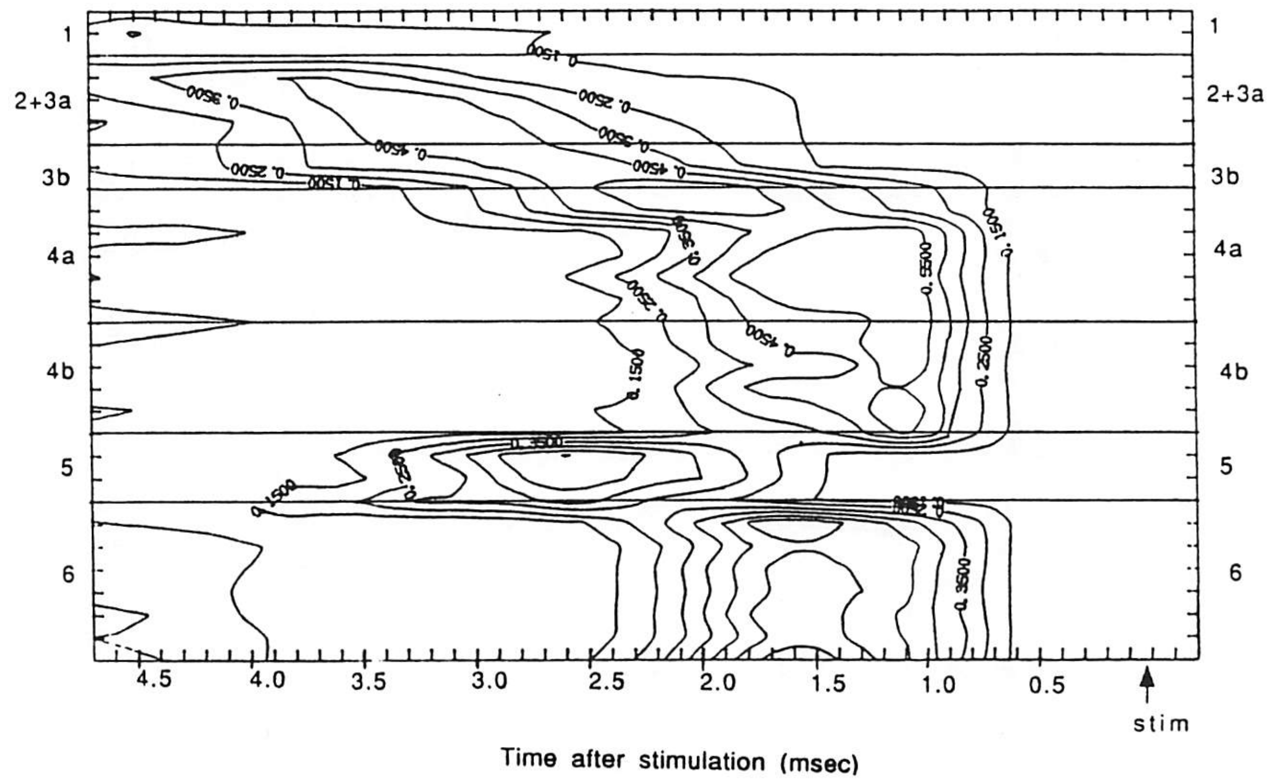

1
2+3a
3b
4a
4b
5
6
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
stim
Time after stimulation (msec)

Fig. 5

MediaView: An Editable Multimedia Publishing System Developed with an Object-Oriented Toolkit

Richard L. Phillips
Los Alamos National Laboratory
rlp@lanl.gov

Abstract

MediaView is an *editable* multimedia publication system, a feature that sets it apart from any system currently or previously available. It achieves power, flexibility, and ease of use through the familiar cut, copy, and paste exemplar normally applied only to text in WYSIWYG word processors. In MediaView all components are manipulated this way. MediaView was developed on a NeXT computer using the object-oriented development system called NeXTstep. The powerful base classes defined there, coupled with Objective-C features like dynamic binding, enabled MediaView to be developed in a short time but with a rich feature set. Two key subclasses of NeXTstep objects give MediaView its strengths. Through them, arbitrarily complex multimedia components can be installed in a document and then copied and pasted, just as if they were words. MediaView has two document formats — one for compactness and speed and one for interchange with other applications and other computers. The latter format is based on a hierarchical file system, which is well suited to UNIX but also to other systems, like that found on a Macintosh.

1. Introduction

MediaView is a multimedia digital publication system that was designed to be flexible and free from restrictions. It was also designed to take maximum advantage of the media-rich hardware and software capabilities of the NeXT [1] computer, including the features of the NeXTdimension [2] sub-system. Since MediaView does not tacitly impose the publisher's agenda on the reader, it is an extremely general system and is free of artificial structure and inconvenient metaphors.

MediaView is easy to use and understand. It is based on the what-you-see-is-what-you-get (WYSIWYG) word processor metaphor, something familiar to most computer users. In addition to text, that metaphor is extended to include all multimedia components. And like text, these components are subject to the select/cut/copy/paste paradigm, making them as simple to manipulate as words. As a result, powerful and complex MediaView documents can be constructed by non-specialists. Also, anything MediaView displays on the screen can be printed, or captured as a PostScript or TIFF file for processing by other systems.

In addition to the expected multimedia components like graphics and audio, MediaView supports several non-traditional components. These include full color images; object-based animations; image-based animations; live video; mathematics; and custom, dynamically loadable components. To provide such a range of capabilities, MediaView fully exploits the platform integration and media richness of NeXT, NeXTstep, and NeXTdimension.

Figure 1 shows the primary MediaView window. The *summary view* is a scrolling list of the names of the documents currently loaded in MediaView. The highlighted item is the document presently being manipulated, which is displayed in the scrollable *content view.*

A complete description of MediaView from the user's perspective will appear in [3]; technical details are emphasized here.

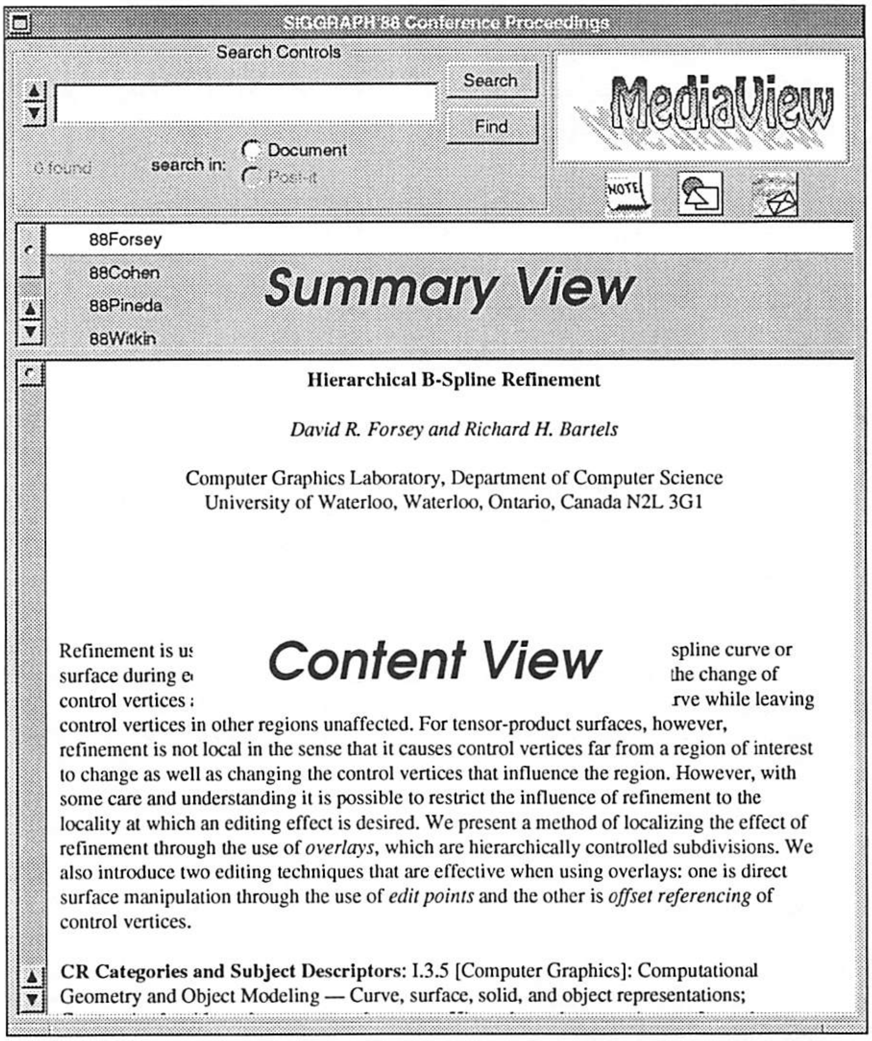


Figure 1. MediaView Main Window

2. The Structure of MediaView

MediaView owes a measure of its capability to the underlying object-oriented NeXTstep environment [4]. The major components of NeXTstep are the Window Manager, the Application Kit and the Interface Builder. The Application Kit was most important to the development of MediaView, closely followed by the Interface Builder.

The Application Kit is a collection of about 50 classes of commonly used graphics user interface objects. The application programmer's interface to the Application Kit is based on Objective-C [5]. The style of programming in NeXTstep is to subclass objects in the Application Kit and then to override default behavior or add new behavior. While one's application-specific code can be written in ordinary C, MediaView has benefitted greatly from being programmed primarily in Objective-C.

Nearly all classes defined in the Application Kit were used in the development of MediaView, but two, Text and View, played a critical role. Figure 2 depicts the entire inheritance hierarchy of the Application Kit. Everything descends from the Object class and inherits its properties. The Responder class, so named because it responds to events, is an important subclass, as evidenced by the fact that most other kit objects

descend from it. One of its descendants is the View class, which provides a structure for drawing. Descended from that is the Text class, whose basic function is to display and edit text. The Text class forms the basis for the WYSIWYG metaphor of MediaView. That means that functions like copy and paste are inherited by its subclasses.

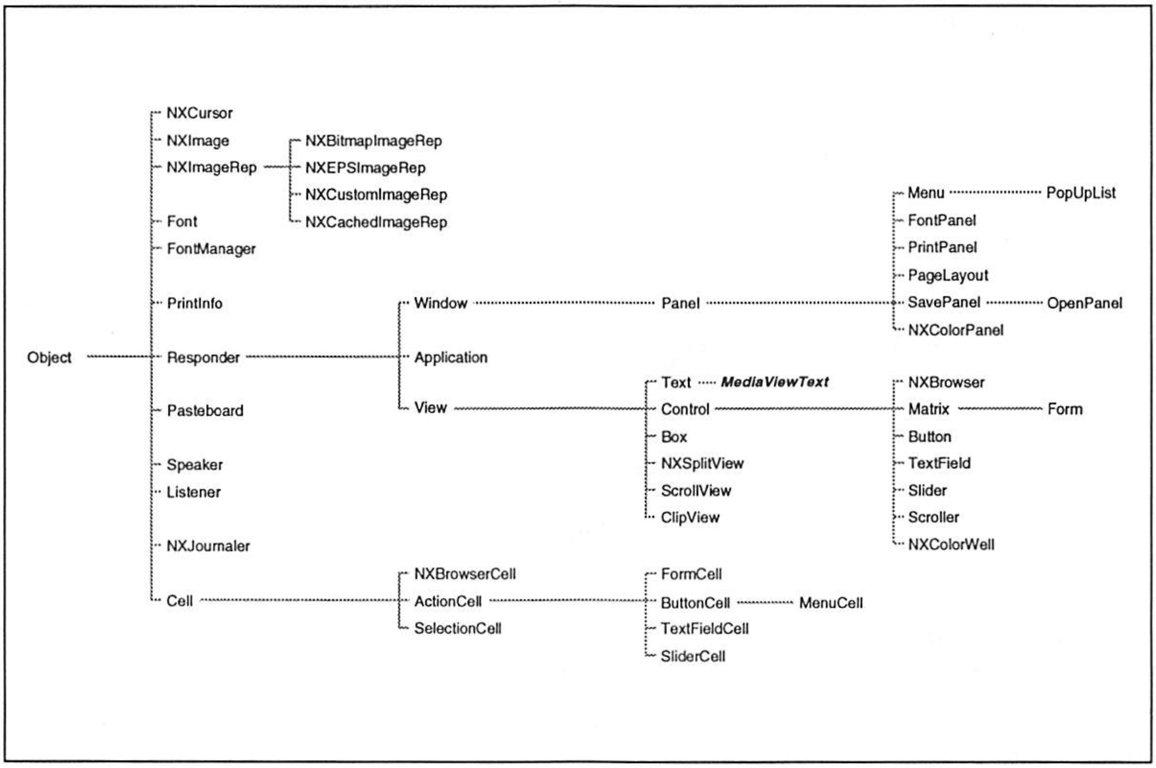


Figure 2. Application Kit Classes

The multimedia components of MediaView are actually subclasses of the View class that are installed in a subclass of Text and appear to the Text object as just another character. It is up to the View subclass to implement whatever behavior is appropriate for that component. Part of that behavior is to properly participate in editing operations, like copy and paste. This requires that they respond to object instance archiving (writing) and unarchiving (reading) methods which are invoked by the parent Text object when the user performs an editing operation. Aside from that, the multimedia components (Views) can be arbitrarily powerful and complex. Some idea of the range of possibilities can be gleaned from Figure 2 by noting the variety of classes that descend from View. Of course, since Text descends from View, components based on Text objects can themselves be embedded in a Text object. This feature is used to produce the digital equivalent of a Post-it note.

2.1. MediaView Subclasses

Installation of multimedia components into a document relies upon the ability to insert subclasses of the Cell class into the data structure of an instance of the Text class. In general, Cells provide rudimentary capabilities for the display of text and icons, but here they merely act as a place holder. It is up to the developer to implement methods for the cell's default behavior, such as how it should be highlighted, react to events and draw itself. More important, the developer must implement all subclasses of the View class that give the multimedia components their distinctive behaviors.

Two fundamental subclasses form the basis for implementation of MediaView documents. They are **MediaViewCell**, a subclass of Cell, and **MediaViewText**, a subclass of Text. The **MediaViewCell** class

provides all the required behaviors of Cell and implements the methods needed for cutting, copying, pasting and writing/reading multimedia components to and from files. Figures 3 and 4 show where these classes fit in the Application Kit hierarchy and some key methods for each subclass.

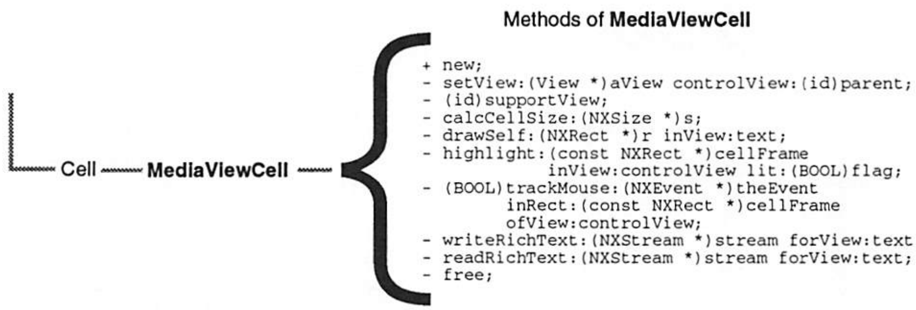


Figure 3. The MediaViewCell Class

The **MediaViewText** class overrides the default delete, cut, copy and paste methods of Text. In addition, it provides a dynamic runtime inventory of the structure of a MediaView document, keeping track of the addition, deletion and movement of multimedia components within the text stream. Also, through this inventory, it is possible to send messages to components that exhibit special behavior. For example, one can instruct an animation component to begin its action when the user scrolls it to a position of visibility in the document window.

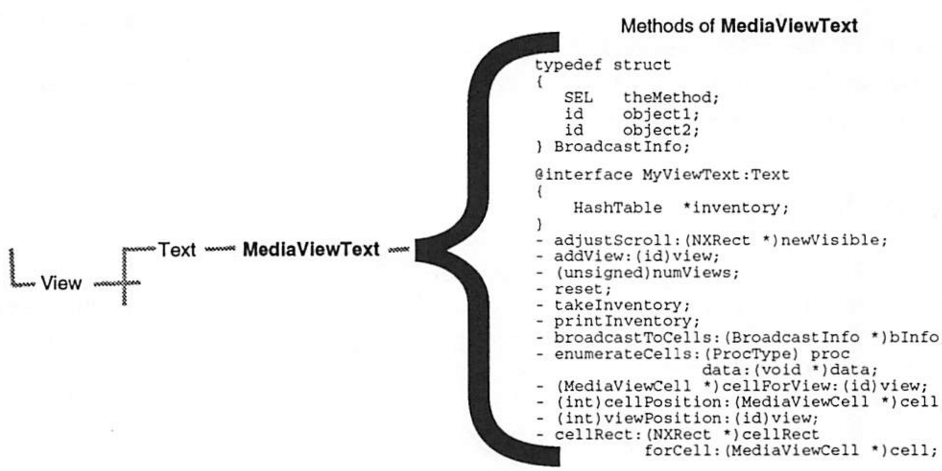


Figure 4. The MediaViewText Class

2.2. MediaView Runtime Data Structure

Actually, one can broadcast any message to the multimedia components to which they can respond. As appropriate, they could be asked to print themselves, play themselves or push their own buttons. The design of the inventorying scheme, updating its dynamic data structure, and associated routines that access it was one of the challenges met in developing MediaView. Briefly, the important components are a HashTable (a common class that is part of the Objective-C environment [6]) for maintaining the runtime data structure and a method for populating it. The hash table entries consist of the id of the **MediaViewCell** and its current integer ordinal position in the text stream. The code for these components is:

```
HashTable  *inventory;
inventory = [[HashTable alloc] initKeyDesc:"@" valueDesc:"i"
                               capacity:0];

- takeInventory
{
    int  r , numChars;
    int  numRuns = theRuns->chunk.used/sizeof(NXRun);

    if( [inventory count] ) [inventory empty];
    numChars = 0;
    for( r=0; r<numRuns; r++ ) {
       numChars += theRuns->runs[r].chars;
       if( theRuns->runs[r].rFlags.graphic ) {
          [inventory insertKey:(const void *)theRuns->runs[r].info
                    value:(const void *)numChars - 1];
       }
    }
    return self;
}
```

The data structure of theRuns can be found in the description of the Text class [6]. Through this infrastructure MediaView objects can determine their current position in the document —

```
- (int)cellPosition:(MediaViewCell *)cell
{
    return (int)[inventory valueForKey:cell];
}
```

and with the aid of this struct —

```
typedef struct
{
   SEL   theMethod;
   id    object1;
   id    object2;
} BroadcastInfo;
```

one can broadcast to all elements via —

```
- broadcastToCells:(BroadcastInfo *)bInfo
{
    [self enumerateCells:passMethod data:bInfo];
    return self;
}
```

.

Other declarations and methods referred to above are:

```
typedef int (*ProcType) (id self, id view,
                        NXSelPt *sp0, NXSelPt *spN, void *data);

- enumerateCells:(ProcType) proc data:(void *)data
{
    NXHashState  state;
    const  void  *key;
    const  void  *value;

    [self takeInventory];
    if( [inventory count] ) {
       state = [inventory initState];
       while( [inventory nextState:&state key:&key value:&value] ) {
          (*proc) (self, key, &sp0, &spN, data);
       }
    }
}

static int passMethod (id self, id cell,
                       NXSelPt *sp0, NXSelPt *spN, void *data)
{
    id              view;
    id              ret;
    SEL             meth;
    id              obj1;
    id              obj2;
    NXRect          viewFrame;
    NXRect          cellFrame;
    BroadcastInfo   *bInfo = data;

    meth = bInfo->theMethod;
    obj1 = bInfo->object1;
    obj2 = bInfo->object2;
    view = [cell supportView];
    if( [view respondsTo:meth] ) {
       [view getFrame:&viewFrame];
       [self cellRect:&cellFrame forCell:cell];
       [view moveTo:NX_X(&cellFrame) :NX_Y(&cellFrame)];
       if( obj2 )
          ret = [view perform:meth with:obj1 with:obj2];
       else
       if( obj1 )
          ret = [view perform:meth with:obj1];
```

```
        else
            ret = [view perform:meth];
    [view moveTo:NX_X(&viewFrame) :NX_Y(&viewFrame)];
    }
}
```

This all demonstrates the power of an object-oriented environment, especially one that supports dynamic binding. The BroadcastInfo struct is assigned a value at a level higher than **MediaViewText** and passed to the **broadcastToCells** method. This struct contains a method descriptor, theMethod, and optionally the ids of two objects that are arguments for the method. For example, to cause all the MediaView methods to print themselves (where appropriate), the method passed in bInfo would be printPSCode. That method requires only one argument so the third field of bInfo will be nil. Then the **enumerateCells** method looks through the HashTable inventory and invokes the passMethod procedure for each component. Then passMethod checks to see if the view belonging to the cell understands the method in bInfo and, if so, sends a message to the view using a method defined in the Object class **perform:**(SEL)*aSelector* **with:***anObject*. This allows messages to be sent that are not determined until run time.

3. Standard Multimedia Components

As mentioned earlier, MediaView provides the user with a wide range of multimedia components. These are described in detail, with examples, in [3]. While many of these components are not found in any other multimedia system, they are considered to be standard in MediaView. In any system, aside from text, one expects to find some provision for sound, line art, video, and sometimes animation. In addition to these, MediaView provides two facilities for animation, accommodates 24 bit/pixel color images, provides a linkage to mathematica via "live" equations, and three kinds of annotations: text, graphics, and sound. For users whose needs are broader Mediaview provides a custom component definition facility, which is described in Section 4. One standard component warrants discussion here. It is a subclass of Button called **MediaViewButton**.

3.1. MediaViewButton, a Versatile Button Subclass

Some of MediaView's standard components make use of a subclass of the Application Kit's Button object called **MediaViewButton**. It is a kind of *hyperbutton* because it does considerably more than the usual Button subclass. What you inherit from Button is its ability to play a sound, display an image, (primary and alternate), and the specification of an action method. There is no default action method; it must be defined by a subclass. **MediaViewButton** is used to display line art, activate "live" equations, and initiate algorithm animations. The user is made aware of the presence of a **MediaViewButton** by a small magnifying glass icon composited into its active area. This means it is no ordinary image, but is inspectable by clicking the mouse within it. Figure 5 is an example of a **MediaViewButton** used to animate a circle scan conversion described in [7].

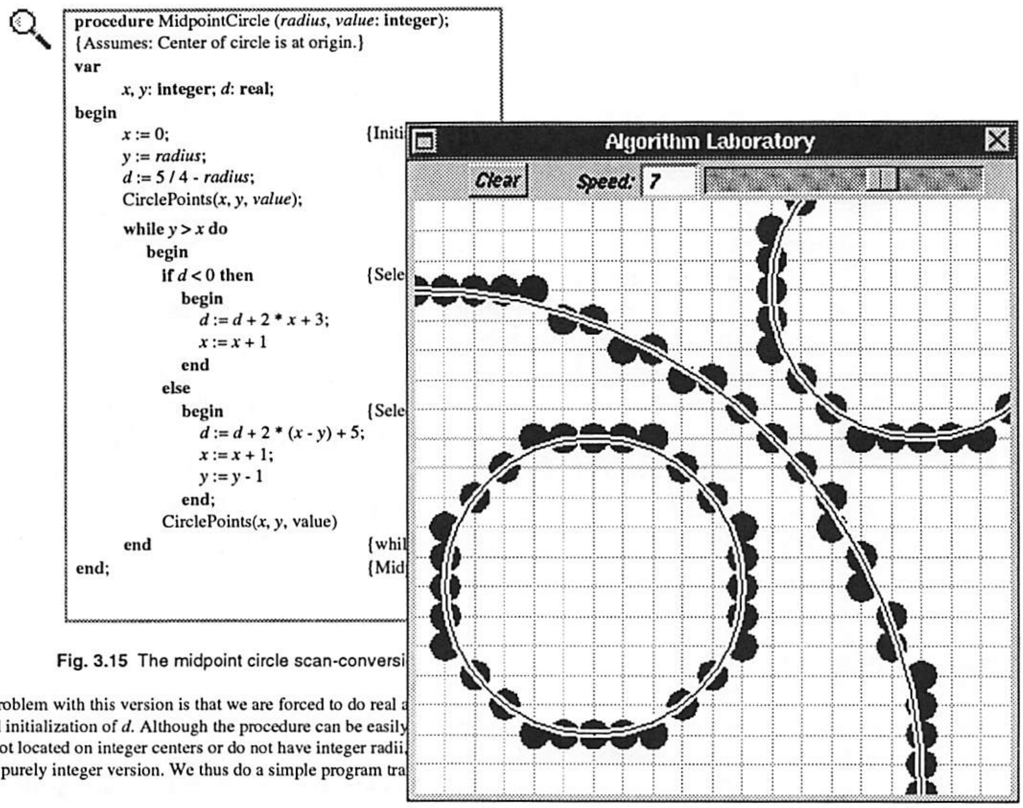


Figure 5. Algorithm Animation

The text of the scan conversion algorithm is all the user normally sees, but when the mouse is clicked within its area, the Algorithm Laboratory window opens. The user can use the mouse to rubber-band an ideal circle and then watch it being approximated by the scan conversion algorithm. The speed of evolution can be adjusted by the slider.

4. Custom Components

In addition to its standard components, MediaView provides a custom component definition capability. A custom component is based on a class which is *a priori* unknown to MediaView; it must be dynamically loaded when it is first referenced. It is not necessary to have access to the source code that defines this class, only an object file or library. This suggests that users of MediaView can develop custom component classes and easily make them available to others. In this way, the power of MediaView can be indefinitely extended.

A developer of a custom component class need insure that the class can be instantiated by a method with the name :

newFromParameterFile:(char *)paramFile.

The file whose name is passed to this method will contain all the information the class needs to initialize instances, and any additional data about sounds and images. How much information is needed, and what the object does with it is transparent to the user and to MediaView. For enhanced portability, it is

even possible to read initialization information from a specially named Mach-O segment. If the developer wishes the class to avail itself of the editing capabilities of MediaView, archiving methods for writing and reading the object must be provided.

When a MediaView author inserts a custom component into a document, the system checks to see if the class is known in the current load module. If it is, it is simply instantiated. If not, it must be loaded from an object file or an archive. This is facilitated by a dynamic, runtime class loading facility which is part of the NeXT operating system support library.

Several custom components have been written to accomplish such things as three dimensional dataset viewing, embedded live video clips, and non-standard animations. A simple example, a bouncing lines animation, is shown in Figure 6.

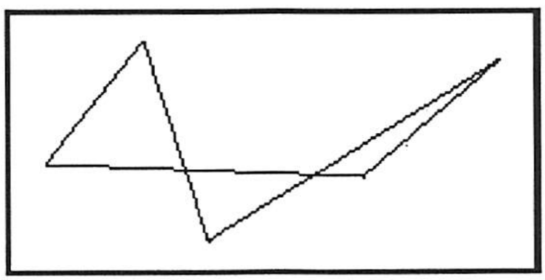


Figure 6. A Simple Custom Component

Some code fragments for this component follow. Shown are the declaration of some instance variables, the instantiation method, the contents of the initialization file, and the archiving methods needed to effect cut, copy, and paste in MediaView.

```
/* Instance Variables */
@interface BouncingLinesView : View
{
  BOOL    running;               /* Whether we are animating */
  BOOL    playing;               /* Whether sound is playing */
  NXRect  drawRect;              /* Frame size */
  int     numCorners;            /* Number of actual corners */
  struct cornerStruct {
    int xLoc, yLoc;
    int xVel, yVel;
  } corners[MAXNUMCORNERS];      /* Corners and velocities for polygon
  char    ops[MAXCORNERS+1];     /* Array of PS operators for user path */
  short   data[MAXCORNERS*2];    /* Array of data points for user path */
  short   boundingBox[4];        /* Bounding box of user path */
  id      viewSound;             /* Sound object */
}

/* Instantiation Method */
@implementation BouncingLinesView
+ newFromParameterFile:(char *)paramFile
{
  FILE    *fp, *fopen();
  NXRect  frm;
  char    *soundFile=malloc(512);
```

```
  fp = fopen(paramFile, "r");
  fscanf(fp, "%f %f %f %f %d", &frm.origin.x, &frm.origin.y,
                                  &frm.size.width, &frm.size.height,
                                  &numCorners);
  self = [super newFrame:&frm];
  NXSetRect(&drawRect,bounds.origin.x,bounds.origin.y,
                      bounds.size.width,bounds.size.height);
  NXInsetRect(&drawRect,2.0,2.0);
  running = NO;
  if( fscanf(fp, "%s", soundFile) != EOF ) {
     viewSound = [Sound newFromSoundfile:soundFile];
     [viewSound setDelegate:self];
     playing = NO;
  }
  fclose(fp);
  return self;
}

/* Contents of Parameter File */
0. 0. 200. 100. 5

/* Archiving Methods */
- write:(NXTypedStream *)stream
{
    [super write:stream];
    NXWriteTypes(stream,"cci",&running,&playing,&numCorners);
    NXWriteRect(stream,&drawRect);
    NXWriteArray(stream,"{iiii}",MAXNUMCORNERS,corners);
    NXWriteArray(stream,"c",MAXNUMCORNERS+1,ops);
    NXWriteArray(stream,"s",MAXNUMCORNERS*2,data);
    NXWriteArray(stream,"s",4,boundingBox);
    NXWriteObject(stream,viewSound);
    return self;
}

- read:(NXTypedStream *)stream
{
    [super read:stream];
    NXReadTypes(stream,"cci",&running,&playing,&numCorners);
    NXReadRect(stream,&drawRect);
    NXReadArray(stream,"{iiii}",MAXNUMCORNERS,corners);
    NXReadArray(stream,"c",MAXNUMCORNERS+1,ops);
    NXReadArray(stream,"s",MAXNUMCORNERS*2,data);
    NXReadArray(stream,"s",4,boundingBox);
    viewSound = NXReadObject(stream);
    return self;
}
```

These archiving methods, write: and read:, are typical in that the superclass is first asked to archive (or de-archive) its structure and then instance variables for the subclass are written (or read). Typed streams, as used in, say, NXWriteArray above, are a standard feature of the NeXTstep environment.

5. Document Construction

There are several ways to construct new MediaView documents. By far the simplest is to use the "drag and drop" paradigm that is available in several graphical user interfaces. For MediaView this means dragging an icon that represents a text file from the NeXT file viewer and releasing it over the summary view. This action converts the file to MediaView internal format, adds its title to the summary view, and displays its text in the content view. Documents that can be processed by MediaView in this way are those produced by WriteNow, FrameMaker, and any file that is in the Rich Text Format (RTF). Several applications can produce RTF documents, including Microsoft Word and WordPerfect.

Once the textual structure has been produced, a user can begin to add multimedia components interactively. Annotations — textual, graphical, or aural — can be added using drag and drop icons, and TIFF or Encapsulated PostScript images can be pasted into the document at the current cursor location. Once finished, the document can be saved, complete with all multimedia components.

It is also possible to construct a document non-interactively, by populating a hierarchical directory with specified text, image, and sound files. This approach is described below.

5.1. Source Document Structure

The most flexible and portable MediaView document representation is a hierarchy of directories and files. Because it is human-readable and human-editable, it is referred to as a "source" representation, analogous to a computer program. The document is rooted in a directory with an extension of *cmbr*, standing for corpus member. Below that is an information file which describes how many of each type of multimedia component is in the documen,t and directories containing data for each type. These directories must be named *animations*, *draw_its*, *eqns*, *images*, *lineart*, *math_notebooks*, *read_its*, *hear_its*, *video*, and *custom*. In the lineart directory, for example, one would find files named Fig1.eps, Fig2.eps, ... , FigN.eps, where N is the number of line art components in the document. A video directory would have files named Command1, Command2, etc. These files could contain seek and play sequences for a videodisc player. As a final example, a custom directory would contain file pairs named (View1, Param1), (View2, Param2), etc. The ViewN files contain the class name of the custom component while the ParamN files contain initialization information for instances of the class.

Constructing a document in this form has the disadvantage of requiring some computer expertise of the user. On the other hand, source-structured documents are highly portable. In fact they can be constructed on many computer systems and transferred to a NeXT only to load it into MediaView. This is accomplished, by the way, by dragging the folder's icon into the summary view. MediaView parses the entire directory hierarchy and converts it to its internal data format. At this point the document is indistinguishable from one that was constructed interactively.

5.2. Binary Document Structure

Currently, a MediaView document is saved to a file with a structure that reflects the runtime data structure of its MediaViewText object. A stream is opened to a file and typed stream descriptions of all multimedia components are inserted into it in the order in which they are encountered. This approach provides for a compact file and fast reading and writing, but at the expense of flexibility.

A new document structure has been designed, but not yet implemented. This is based on BTree files and will enable MediaView documents to be "smart" documents. One example of the advantage of this approach is dynamic merging of search indices of the textual part of several documents. Currently, in order to do a rapid search for a text pattern that several documents have in common, all candidate documents

must be indexed at once. The BTree structured document will contain only its own indices and will be merged with those of other documents upon demand.

6. Future Enhancements

The most obvious and most important enhancement is a hyperlinking capability. This has been designed and will be implemented in the next few months. Its design draws upon the rich NeXT development environment, in particular the suite of BTree classes that are available. The database that maintains linking information will use the features of those classes. Hyperlinking will be accomplished with an overlay strategy, that is, linking information will not be part of the document. This enables different users to have different linking strategies without having to duplicate document data.

Finally, work is underway to provide more powerful authoring tools for the most sophisticated MediaView components. This will allow, for example, an author to take full advantage of the capabilities of dynamically loadable custom objects. Among other things, this tool will allow an author to edit segments in the Mach-O object files that describe the characteristics of the component.

7. References

1. Clapp, D. *The NeXT Bible: Hardware and Software Systems for the NeXT Computer*, Brady, New York, 1990.

2. NeXT Computer, Inc., NeXTdimension, Product No. N6030, September,1990, Redwood City, CA.

3. Phillips, R. L., MediaView: A General Multimedia Digital Publication System, to appear in *Communications of the ACM*, July, 1991.

4. Thompson, Tom, The NeXT Step, *Byte Magazine*, Vol. 14, No. 3, March, 1989, p. 265.

5. Cox, B. J., *Object Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading, MA, 1986.

6. NeXT Computer, Inc. NeXTstep Reference, Product No. N6007B, December, 1990, Redwood City, CA.

7. Foley, J. D., van Dam, A., Feiner, S. K., Hughes, J. F., *Computer Graphics: Principles and Practice*, Addison-Wesley Publishing Co., Reading, MA, 1990.

Dick Phillips is a Staff Member in the Computer Graphics Group, Computing and Communications Division of the Los Alamos National Laboratory. His current research interests are scientific visualization, multi-media workstations, distributed computing, window systems, and digital publication. Prior to coming to Los Alamos, he was a Professor of Electrical & Computer Engineering and Aerospace Engineering at the University of Michigan for over 20 years. There he was instrumental in establishing a several hundred node workstation network for student and faculty use in the College of Engineering.

He received a BSE in mathematics in 1956, a MSE in aerospace engineering in 1957, and a PhD in aerospace engineering in 1964, all from the University of Michigan.

Phillips is a member of IEEE and ACM.

His address is Los Alamos National Laboratory, P. O. Box 1663, MS B-272, Los Alamos, NM 87545.

A Structure for Transportable, Dynamic Multimedia Documents

Dick C.A. Bulterman (dcab@cwi.nl)
Guido van Rossum (guido@cwi.nl)
Robert van Liere (robertl@cwi.nl)

CWI: Centrum voor Wiskunde en Informatica
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Abstract

This paper presents a document structure for describing transportable, dynamic multimedia documents. Multimedia documents consist of a set of discrete data components that are joined together in time and space to present a user (or reader) with a single coordinated whole. Transportable documents are those in which the document structure can be accessed across system environments independently of individual component input or output dependencies; dynamic documents are those in which the synchronization of document components are not staticly defined as an integral part of the data definition but are dynamicly defined as attributes of the general document structure.

The focus of this paper is the presentation of the basic building blocks of the CWI Multimedia Interchange Format (CMIF). CMIF is used to describe the temporal and structural relationships that exist in multimedia documents. In order to put our work in a concrete context, we start our discussion with a brief description of the portability requirements for documents used within the CWI/Multimedia Pipeline. We then provide a layered description of our document structure format; this format provides a means for expressing a document in terms of *synchronization channels*, *event descriptors*, *data descriptors*, *data blocks* and *synchronization arcs*, each element of which contains a set of appropriate descriptive attributes. The paper describes each of these concepts abstractly as well as in the context of a uniform example. The paper concludes with a discussion of our intended future direction in using the various attribute descriptors to control a broad range of activities within the CWI/Multimedia Pipeline.

1. Introduction

A visitor to any computer-industry trade show is immediately confronted with the state-of-the-art of multimedia systems. Even modest personal systems demonstrate an impressive ability to simultaneously manipulate a variety of (chiefly output) media in a way that provides a dazzling display of technological cleverness and audio/video wizardry. Birds flying across medium-resolution color screens can be frozen in mid-air, then cut out of their environment and pasted on top of a composite background with remarkable ease; images can be video mixed, then translated, rotated and scaled at a speed that once was impossible with even the highest-performing (and highest costing!) workstations. Novice users can take information from CD-ROMs that contain data of several media, giving promise to better teaching tools for children or clearer maintenance and repair guides for automobile mechanics. At first glance, current generation multimedia systems can do nearly everything that a user could hope for, with the implied promise that even not-yet hoped for things are just around the technological corner.

In reality, however, there are several major problems that confront the user of multimedia systems. One problem is that the elements being manipulated within multimedia systems consist of raw data rather than structured information; the bird flying in the example above usually is little more than a sequenced video FAX that has a representation but little inherent meaning. This can limit the amount of intelligent processing that can ultimately take place by application programs or support hardware. A second problem is that the representation and manipulation of data is highly machine and/or device dependent. This means that information can not easily be shared

among different types of systems or devices. A third problem is that the synchronization present within multimedia applications is often implicitly encoded as a function of the speed of a particular system and interface, limiting the ways that interaction among elements can be expressed and (ultimately) implemented. The result of these problems is that it is often difficult to share documents among different applications or across similar applications that are implemented on different computing platforms. Even in those systems that allow sharing mechanisms, the fact that information on data and data formats is often intimately entangled with details of presentation timing or with details of presentation formats, screen resolutions, window placements, etc., makes it difficult for higher-level authoring tools to effectively coordinate the synchronization of a number of otherwise autonomous data streams and for manipulation tools to effectively separate data presentation/placement from data information encodings.

This paper presents a method for addressing the three problems above by focusing on the manner in which information in a multimedia application is represented; we then consider ways of using that representation to coordinate the manipulation of the composite parts of the application itself. In order to do this, we define CMIF, the CWI Multimedia Interchange Format [Rossum91]. CMIF encodes a multimedia document as a collection of data of potentially diverse media *and* as a set of structure and synchronization relationships that describe how the data components are to be presented and manipulated. CMIF has been motivated by two goals: first, to define a description that separates the temporal, spatial, and content-based aspects of multimedia documents, and second, to investigate means of using the document description rather than the data contents to control the interrogation and synchronization of one or more document sets. It was developed to provide the ''glue'' that binds together various components of the CWI/Multimedia Pipeline; this pipeline is briefly discussed below.

It should be noted that while the purpose of CMIF is to provide a means for expressing dynamic relationships within a document in a transportable manner, this does not imply that all systems that recognize the format will also be able to implement a particular document. (It would be impossible to implement the flying bird document from above if the target system had no display, for example.) What CMIF *can* provide, however, is a structured basis upon which a given system can determine whether it can support the requested document or not.

In the sections below, we describe CMIF in terms of its abstract properties and in terms of an example multimedia document. To motivate the particular requirements of our work, we start with a brief description of the CWI/Multimedia project. We then introduce the basic building blocks used within CMIF. Next, we describe our multimedia example, and use it as a way of exploring some of the more detailed aspects of the document format. This is followed by a more detailed look at CMIF, with special attention provided to the question of document synchronization. We conclude with a brief discussion of some of the ways in which a document structure can be used to manipulate multimedia documents without accessing the (potentially huge) multimedia data sets themselves.

2. The CWI/Multimedia Project

The CWI/Multimedia project is a new area of coordinated research that provides researchers in the areas of systems architecture, operating systems, distributed databases, user interface design and interactive systems to consider problems associated with the manipulation of multimedia data in a distributed computing environment. Areas of interest on multimedia systems can be described in terms of the collection of phases that exist during the lifetime of a multimedia document; these phases have been grouped together into the CWI/Multimedia Pipeline. Figure 1 gives a schematic of the components of this Pipeline.

The elements of the pipeline are:

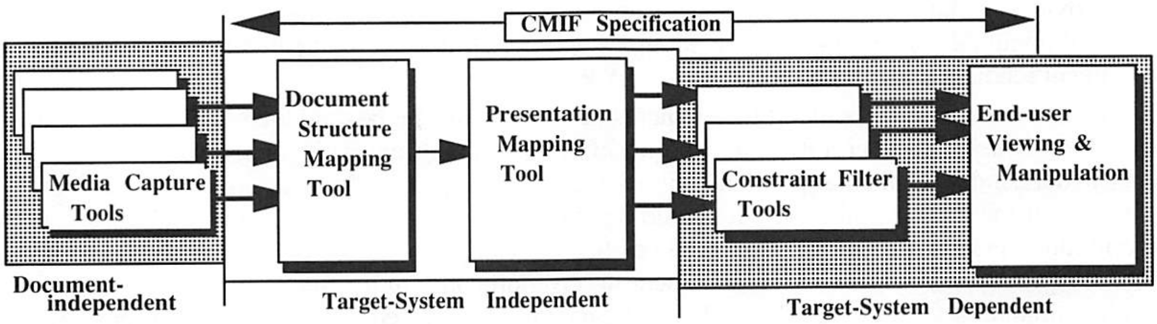


Figure 1: Schematic of the CWI/Multimedia Pipeline

- *Media Block Capture Tools:* a set of tools that will allow the user to iteratively capture (and edit) the atomic pieces of information that will be included in a composite document. In general, our concern is not with the hardware technology associated with the capture of a particular medium: we expect that equipment vendors or third-party organizations will do this better than we can. Instead, our focus is on providing descriptive tools that allow higher-level processing of various bits of collected information. (This processing may include scheduling, searching, comparing, editing, etc.) Note that the concept of an atomic item still allows for a complex structure; since the goal of these subsystems is chiefly to compile descriptors, a broad range of underlying systems can be supported.
- *Document Structure Mapping Tool:* this tool allows the user to express relationships among individual media blocks. The relationships are primarily temporal and spatial. Information from this tool is used by later components to support presentation, local system filtering and editing/reading of a multimedia document. The document structure mapping tool produces a document in the CMIF format.
- *Presentation Mapping Tool:* this tool allows portions of a document to be allocated to a *virtual presentation environment*. This tool is used to allocate virtual presentation ''real estate'' (such as areas on a display or channels of a loudspeaker) to a given multimedia document. Some of the mapping information may come from ''preference'' defaults provided with each atomic media block, or they may need to be added by this tool. In either case, this tool manipulates the definitions provided in the CMIF document and creates a presentation map that can be manipulated separately from the document itself.
- *Constraint Filtering Tools:* these tools allows the end-user presentation system to filter components of the document to meet local processing constraints. (This corresponds to a mapping of the document from the virtual presentation environment to a physical presentation environment.) Typical filterings may include 24-bit color to 8-bit color, color to monochrome, high-resolution to low resolution, full-frame-rate video to sub-sampled rate video, etc. As with all components, the assumption is that this tool *manages* a constraint mapping; the actual constraint implementation will be supported by user level, operating system, or hardware level modules.
- *Document Viewing and Reading Tools:* These tools present a document (based on the document structure map, the presentation map, and the local filter map) and provide a means for a reader to ''view'' or (possibly) edit a document. Note that the document structure map

provides a data-independent, position-independent and system-independent view of the multimedia document being read, acting as an internal table-of-contents function for subsequent reading and editing tools.

The detailed description of each of the elements of the pipeline are beyond the scope of this document. As of this writing, prototype designs of different elements are being undertaken in order to better understand fundamental problems during the processing of multimedia information. From the nature of the pipeline, however, it should be evident that the provision of a central document description is essential if information is to be shared cleanly among disjoint manipulation tools. It should also be evident that a rich document description can improve the performance of document manipulation tools by providing the summary information required by the virtual presentation and constraint tools without requiring the data itself to be manipulated. We return to this point below.

3. Introduction to the CMIF Structure

The purpose of CMIF is to provide a mechanism for describing the structural components that exist in a multimedia document and to describe the synchronization relationships among those structural components. This section provides a first look at the elements of CMIF and then relates it to other document structures used with a multimedia system.

3.1. Overview of the CMIF Building Blocks

The basic CMIF building blocks are summarized in the following table:

Building Block	*Function*
Data Blocks	The basic atomic element of single-media data
Data Descriptors	A set of attributes describing the semantics of the data block
Event Descriptors	A set of attributes describing the presentation of a data block
Synchronization Channels	A placement framework for sequential and parallel events
Synchronization Arcs	The specification of the interaction constraints among events

The relationships among data blocks, data descriptors and event descriptors are given in figure 2.

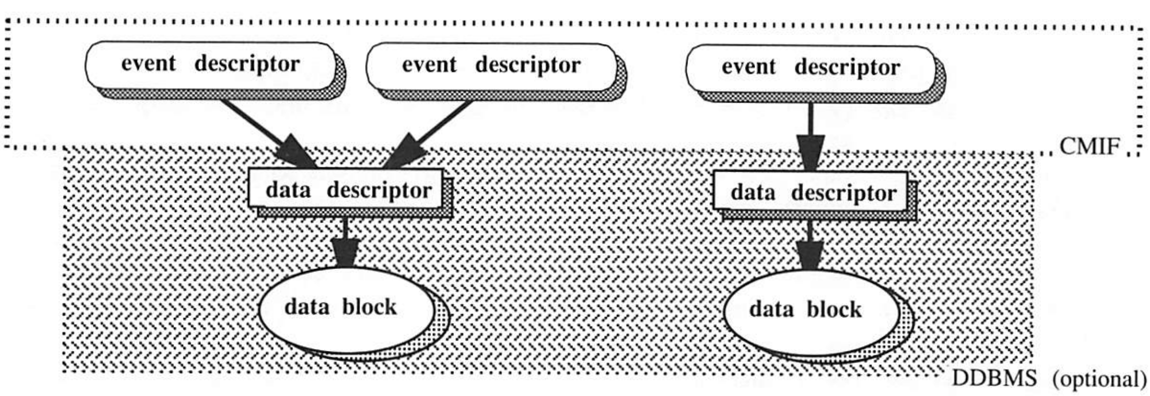


Figure 2: Data blocks, data descriptors and event descriptors

Data blocks contain data that is typically associated with a single medium. Examples may be sound clips, video segments, text blocks, graphics images, etc. They may also be programs that produce information of a particular type. (An example might be a graphics program that produces a rendered 3-D image.) The fundamental property that a data block has is atomicity: it is assumed that, for the purpose of a CMIF-based document, each data block can not be further decomposed or sub-scheduled.

Data block descriptors are collections of attributes that describe the nature of the data block. The CMIF format makes only minimal assumptions about the types of attributes that can be defined for a given block. It does this because it does not interpret the meaning of these attributes—it simply allows them to be passed on to the required system tools that are used to manipulate a multimedia document. Example attributes may be structure information on the data block (its format, its resolution, its length, the resources required to support it, etc.) Note that a database management system may be used to locate and access various data blocks based on the attributes in the data descriptors. This possibility is illustrated by the shaded region in the diagram.

Event descriptors provide a collection of attributes that describe how a single instance of a data block is integrated into a multimedia document. The attributes in the event descriptors allow synchronization information to be expressed in the document. They also provide a means for describing that subset of data descriptor attributes that are required for the efficient access and manipulation of the data block. Note that the primary difference between the data descriptor and the event descriptor is that the event descriptor can be used to define multiple uses of a single data descriptor. Each of these items are discussed further in section 4.

A CMIF description consists of the mapping of event descriptors onto one of a set of *synchronization channels*. Each channel describes how data of a single medium is manipulated in the document. It is possible to have several channels of the same medium type; all data of a type may also be placed on a single channel. The principal role of the channel is to provide a mechanism for event synchronization. Events that are placed on a single channel are synchronized in linear time order, with the start of the second of two events occurring at a (possibly constrainted) time after the completion of the first. Two events that are placed on separate channels may be executed in parallel, either simultaneously or at a (possible constrained) time interval offset relative to each other.

Synchronization information is encoded in terms of *synchronization arcs*. Each arc is a directed connection between two event descriptors, under the convention that the arc is drawn from the controlling event to the controlled event. Each arc has a set of synchronization attributed associated with it; these attributes indicate if the synchronization is strict or approximate. It also allows for the expression of synchronization ranges so as to account for different implementation environments. Synchronization arcs can be placed at the beginning of an event or at the end of the event. If detailed synchronization is not required, then the synchronization arc can be omitted from the description. In this case, event blocks on a channel follow one-another in some system convenient manner and events on different channels have an implied synchronization that is derived from their relative placement.

Figure 3 gives a schematic view of the document structure framework. This view is similar to one that would be expected in a document structure editor tool, although the view here has been simplified for purposes of illustration.

The discussion in this section is intended to provide an introduction to CMIF. Before presenting a more detailed view of the format, we first relate the overall goals of our work to other research in this area. We then present a short description of a simple example that can be used as the basis for a second look at CMIF.

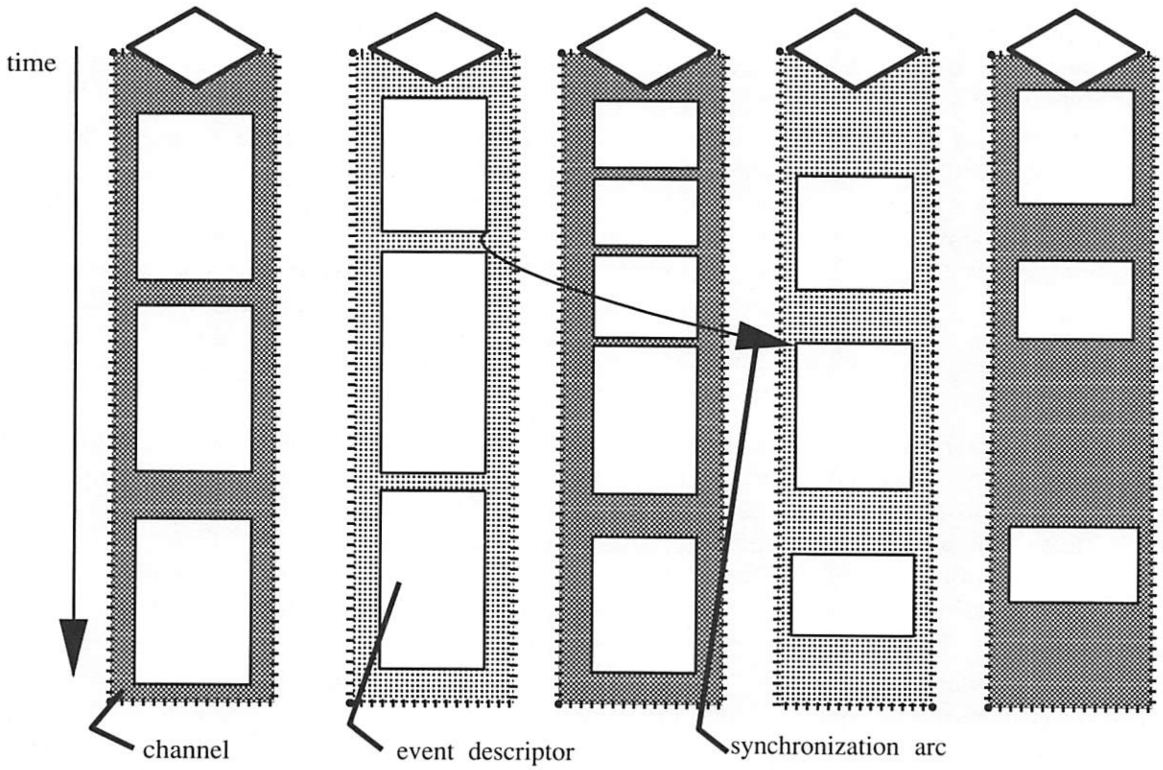


Fig. 3: Document structure components.

3.2. Relationship of CMIF to Other Formats

The main contribution of CMIF is that it provides an explicit means of describing the synchronization information in a multimedia document. It also allows for a clean separation of the various types of attributes required to differentiate the nature of a document from the definition of its components. CMIF is similar in purpose to a number of other document formats, although it differs from some of them in fundamental ways. One example is the Diamond project [Thomas85], where the use of a document structure is limited to the expression of textual and graphical data without explicit time constraints. Another example is Muse [Hodges89], where a time line concept is employed for synchronization, although Muse uses this solely to describe video sequences. There are also commercial document structure formats, such as the MIF format of Frame Technologies' FrameMaker [Frame89]; in general, these formats (while broadly defined) address only the issue of document structure rather than the specification of interaction and interaction constraints. Other formats, such as those for encoding commercial video or audio data, typically do not provide a method for coordinating different data types. This is also the case with research-oriented data descriptions (such as the image transfer format used by Dean, et al [Dean90]) for particular applications areas. Each of these formats are useful *within* a document, however; it is possible that a data descriptor attribute list may include a data encoding field that allows for the specification of a data block in a well-accepted format. This practice is encouraged even though the formats themselves are orthogonal with respect to each other.

Another point of comparison is the relation of our structure to that used in hypertext or hypermedia systems [Halasz90, Yankelovich89]. The entire question of hyper access to data is intimately related to the concepts of document presentation synchronization. Unfortunately, since hyper navigation implies a non-linear ordering of data, it is difficult to directly express all possible synchronization paths that can potentially emanate from a particular event descriptor. One approach that seems to address this problem in an interesting fashion is presented in HyTime [Dean90]; HyTime is a hypermedia document description language that includes the notions of time constraints along with hyperlinks for data. While we suspect that this general problem can be addressed via the definition of conditional synchronization arcs that point to events on separate channels, we have not developed these ideas in sufficient detail to discuss them here.

4. An Example: The Evening News

As an example multimedia document, consider a (pre-created) version of the evening television news. The structure of this document is relatively straight-forward: the news is divided into a number of separate program blocks, each of which consists of spoken text, a main video stream (showing the announcer's head or the usual dramatic on-location scenes), one view of a static background graphic illustration, and one labelling text stream (for identifying the composite screen image). If we further assume that a text-string is synchronized with the presentation for providing either multi-lingual broadcasts or captioning for the hearing impaired, then we begin to see how a dynamic multimedia document can be constructed for a collection of synchronized components.[1]

We find the evening news an interesting sample document for several reasons. First, it is an example that is broadly familiar; while individual customs may dictate order and content, the general structure of a TV news broadcast hardly varies from one country to another. Second, the contents of a news broadcast provides a collection of interesting synchronization problems that can be used to illustrate properties of the CMIF format. Examples of this synchronization include start synchronization across all blocks at the beginning of a story, block synchronization between the video and audio channels, offset synchronization between the graphic channel (where a map or an illustration may be placed) and the audio portion of the news, block synchronization between the visual blocks and the caption text, and synchronization between the placement of a label block and several of the other channels in the application. Note that we will define in more detail how synchronization is specified in CMIF later in this paper; this section is simply used to introduce the general concepts.

Figure 4 contains two views of one program block in the news. Assume that the image is a fragment out of a report on paintings that were stolen from a local museum. Fig. 4a shows a possible TV image and figure 4b provides a high-level view of the structure of a program block document. The broadcast in 4a is divided into five channels: one for the main video stream (showing the reporter presenting the story), one for the sound stream (shown as coming from the side of the display), one for the graphic frame (in this case, showing an image of the painting just stolen), one for the labelling frame (used to identify the story for those just tuning in) and one for the captioned-text string (in this case, presenting an English translation of the Dutch text coming through the speakers). The structure view of the document in 4b consists of blocks that identify the various events in the document. It further consists of a number of data blocks and several timing arcs which provide synchronization information. Note that, under normal circumstances, time moves from the top of each block toward the bottom and that blocks are either explicitly synchronized via timing arcs or implicitly synchronized by virtue of their relative placement.

[1] We include this example for purposes of illustration only. For an application system that address this issue more fully, see [Hoffert91].

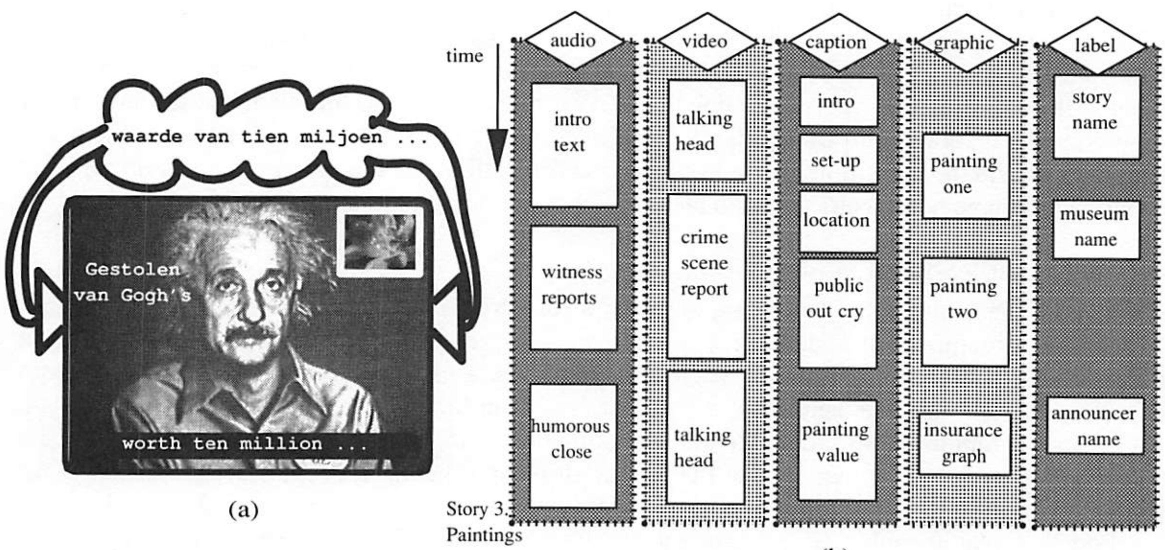


Figure 4: The Evening News as a document (4a) and as a CMIF template (4b).

For purposes of our example, we impose a few restrictions on the news: first, all data fragments are assumed to be pre-formatted. (That is, there is no notion of a spontaneous discussion among the data components and—perhaps more importantly—the length of each of the segments is known in advance.) Note that this restriction is not inherent to the document; we make it for purposes of simplification only. Our second restriction is that the synchronization among blocks is fully described by the components of the document structure. While it is possible to alter the rate of presentation (such as freeze-framing or using slow-motion), it is not possible to alter the order of events within the document by viewing it—re-ordering requires re-editing the document.

In an actual news example, each of the data blocks would need to be created via a creation tool (one for each medium) and several other tools need to exist to handle the allocation of resources to the application. We will disregard these activities for the moment, although we will return to them later in the paper. (An example of the tools necessary can be found in our description of the CWI/Multimedia Pipeline in section 2.)

In the following sections, we return to the basic building blocks of our document structure. We will use the example described in this section to illustrate the nature of each building block and to discuss synchronization issues in the document structure.

5. CMIF: A Second Look

Up to this point, we have described a document in CMIF format as consisting of a collection of data blocks that are accessed via a set of data descriptors. Each of the data descriptors contains the general attributes that describe the data in a manner consistent with the requirements of the users of that data. An event descriptor describes one particular use of a data block. Event descriptors must therefore contain information specific to a particular instance of the occurrence of a data item, including its relationship with synchronization information.

While this general description is correct, it does not fully describe how a multimedia document is structured internally by CMIF. A more complete description starts with the realization that very little of the information described within CMIF concerns actual multimedia data. Instead, CMIF defines a document tree that is used to encode the hierarchical and peer relationships among document events. The tree is a human-readable document that can be passed from one location to another with or without the underlying data. It can be analyzed by the various tools described in section 2 efficiently before any processing of information starts.

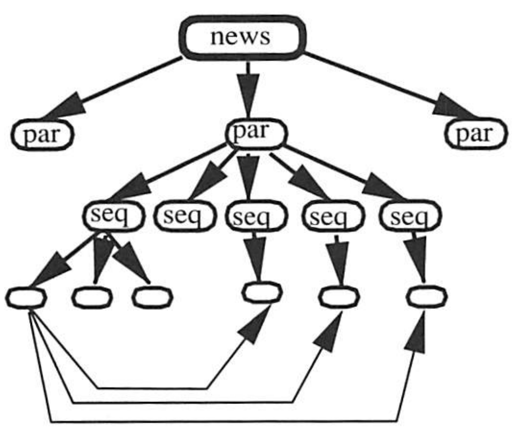
The CMIF tree can be represented as a conventional collection of nodes and branches or it can be represented as an embedded structure. Figure 5 provides a representation of these two views. Each tree consists of a collection of nodes, attributes and synchronization information. Each of these items will be considered in the following sections.

5.1. CMIF Nodes

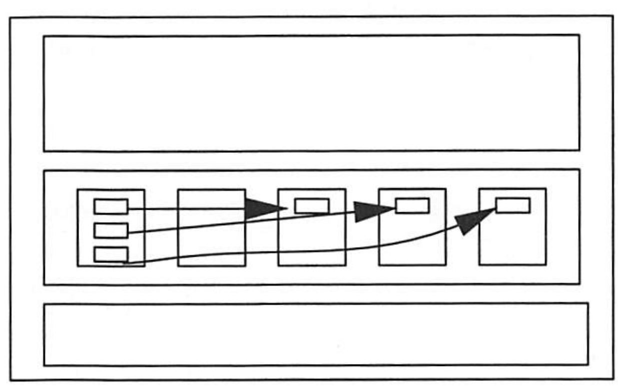
At the root of the tree is a general node that describes the summary structure of a document. This node points to other nodes, each of which in turn point to still other nodes, until finally a leaf is reached that contains a pointer to a data block. The root node has a special function in the tree because it is a place where various directory attributes are found (see below) and because it provides an implied timing reference point for all other nodes in the document.

Each node in the tree can be one of four types:

- *Sequential Node:* each of the child nodes emanating from a sequential node is executed sequentially in a left-to-right order. The children may themselves be either data or complex nodes (that is, either leaf nodes or other sequential or parallel nodes). The parameters of sequential synchronization may be implicitly or explicitly defined. (See below.)
- *Parallel Node:* each of the child nodes emanating from a parallel node is executed in parallel with all of the other children of this node. The children may themselves be either data or complex nodes. The parameters of parallel synchronization may be implicitly or explicitly defined. (See below.)



(a)



(b)

Figure 5: The CMIF tree in conventional (a) and embedded (b) forms.

- *External Node:* this is a leaf node that points to a data descriptor (and thus to an external data block). External nodes should have (or inherit) a file attribute specifying the data descriptor containing the data. If a *slice* attribute is present (see below) only the indicated part of the file is used. Separate *clip* or *crop* attributes may specify a further restriction of the data. The external node provides an indirect reference to data, allowing the structure of the data to be described separately from the structure of the document.
- *Immediate Node:* this is a leaf node containing data rather than a pointer to a data descriptor. The data is either text (the default) or another medium, as indicated by attributes associated with the node. This node is useful for encoding small amounts of data directly in a document *or* for transporting (large amounts of) data across environments that have no common storage server.

The general format of the four nodes is illustrated in figure 6.

In terms of our TV news example, the document tree may contain a collection of sequential nodes, each of which represents one story in the broadcast. (In some cases, commercial inserts or other transition material may also be placed in the document.) Each of the stories may consist of a number of internal components, some of which may be presented sequentially (such as an introduction or a transition), while others may consist of parallel nodes. Eventually, all of the nodes will point to some (multimedia) data that needs to be presented. This data will typically be described by an external node, but it may also contain immediate data (for example, for use with label text).

5.2. CMIF Attributes

Each of the attribute fields in the node contains a pointer to a list of attribute definitions. These definitions generally contain an attribute name, followed by an attribute value. The value field depends on the particular attribute type; clearly, not all attributes will be present in all nodes. Four example attribute value definitions are:

- *ID*, which contains a character value (without embedded spaces) for the attribute;
- *NUMBER*, which contains a numeric value for the attribute;
- *STRING*, which gives a character-string (in quotes, possibly with embedded spaces) value for the attribute; and
- *value** field, which provides a (set of) pointer(s) to other attributes.

seqnode	*attribute_list*	(node →)

parnode	*attribute_list*	(node →)

immnode	*attribute_list*	(data)*

extnode	*attribute_list*	(data_descriptor →)

Figure 6: CMIF node general formats.

One requirement of attribute lists is that each name may occur at most once in each list for each node.

In general, a node can have arbitrary attributes, although for some attributes a standard meaning and format is defined. For some (groups of) attributes, a global consistency rule exists which further restricts their format and/or placement. Some attributes are allowed on all nodes; others are allowed only on certain node types or when combined with other attributes. Some attributes set properties that are ''inherited'' by children (and arbitrary levels of grandchildren) of the node on which they are set unless explicitly overridden; others only affect the node on which they are present. Finally, there is one attribute, ''style,'' which is a shorthand for placing a set of attributes on a node. A representative list of standard attributes in given in the table in figure 7.

5.3. CMIF Synchronization

The ability to describe the interactions among events is of fundamental importance within a multimedia document. While it is obvious that the begin/end relationships among events is important, it is equally important to be able to specify tolerances within timing relationships if a transportable document is to be constructed. To this end, CMIF provides a mechanism for specifying several classes of synchronization primitives within a document. The synchronization information is usually implied rather than explicit, although a facility for increasing the granularity of timing relationships is provided through the synchronization timing arcs.

We divide our discussion on synchronization over four subsections. First, we provide a description of the basic synchronization concerns supported in CMIF documents. We then describe how these can be specified in terms of synchronization arcs and synchronization attributes. Next, we discuss the issues of synchronization conflicts: what can go wrong in a document. Finally, we relate our entire synchronization discussion to the TV News.

5.3.1. General synchronization concerns.

The basic tree structure of CMIF documents imposes a default synchronization that is based on the node type of the ancestors of a data (leaf) node. Within a sequential node, a default synchronization arc exists from the starting node of the arc to its sequentially first child. There are also arcs from the end of leaf nodes to the start of the successor leaf. Finally, an arc exists from the last child of a sequential node to the end of its parent. Parallel nodes have default arcs from the parallel parent node to each of the children of that parent. Similarly, synchronization arcs also exist from the end of each of the children to the end of the parent. In a sequential node, the synchronization relationship between the source and destination of the arc is simply a ''start the successor as soon as possible'' relation. In a parallel node, the relationship across the arcs is a ''start the successor when the slowest parallel node finishes.''

The provision of explicit synchronization arcs gives a document authoring system fine-grain control over the relationships between source and target nodes. As we will see, explicit synchronization arcs allow for bounded delay times relative to a global reference point, as well as an ability to define offsets from the source at which activation can be scheduled to take place. Synchronization arcs also provide the ability to describe either a rigid or a relaxed synchronization constraint—this is especially useful for documents that need to run on diverse sets of hardware.

An explicit synchronization arc can be considered to consist of three basic elements: *reference times, minimum acceptable delays,* and *maximum tolerable delays.* Each is defined as follows:

- *Reference time:* a relative or absolute reference definition for event synchronization. Absolute reference times are specified relative to the root of the document; relative reference times are specified relative to the start or end of a controlling event. Note that a reference time that is at a relative offset of zero from the start of end of the controlling event typically

Attribute	Description
Name	This attribute assigns a name to the current node. Names are optional, and relative to their parent: no two (direct) children of the same parent may have the same name, but otherwise a name may occur more than once in the tree. Names are used by synchronization arcs to reference their source and destination nodes.
Style Dictionary	This attribute defines one or more new styles. It should currently only occur on the root node. The attribute consists of a nested set of names that identify the styles for later reference by *style* attributes. Style definitions may refer to other style definitions as long as no style refers to itself, directly or indirectly.
Style	This attribute specifies one or more styles to be applied to the current node. At runtime, each style name is looked up in the style directory of the root node.
Channel Dictionary	This attribute defines one or more synchronization channels. It should currently only occur on the root node. The names identify channels for later reference by channel attributes. Each channel definition defines the medium used by that channel.
Channel	This attribute specifies to which channel the current node's data should be directed. The name should name one of the channels defined in the root node's channel list. This attribute is inherited by children unless explicitly overridden.
File	This attribute specifies the file to be used by external nodes. It is inherited, so that multiple external nodes can refer to subsections of the same file. It identifies the data descriptor used to reference data; the data may be accessed indirectly via a database system.
T_Formatting	This attribute contains a shorthand list of various formatting parameters that specify how text material will be sent to the text formatting channel. Examples are: font, size, indent, and vspace. Note: it is wise not to use these attributes directly but to place them in a style definition; they are included here for use in exceptional processing situations.
Slice	This attribute specifies a subsection of the file to be used by an external node specifying binary data.
Crop	This attribute provides a mechanism to specify a subimage of an image.
Clip	This attribute provides a mechanism to specify a part of a sound fragment.

Figure 7: Attribute examples.

indicates a coincident parallel relationship (that is, things start or end at the same time). It usually not possible to specify a timing relationship in which the destination starts before the source, although this might be possible to a limited degree if an implementation environment supports pre-fetching and pre-scheduling of events.

- *Minimum acceptable delay time:* a period (possibly zero) that specifies the minimum delay that can be allowed in the synchronization relationship. A minimum delay of 0 units indicates a hard synchronization relationship. A negative delay represents the ability to start the target node sooner than the indicated reference time. A positive delay has no meaning.
- *Maximum tolerable delay time:* a period (possibly infinite) that specifies the maximum delay that can be allowed in the synchronization relationship. A maximum delay of 0 units indicates a hard synchronization relationship. A positive delay gives an upper bound on the permissible delay in starting an event relative to the reference time. A negative delay has no meaning.

The notion of delay times introduces flexibility in the ability to schedule nodes; this is important for transporting documents across different implementation environments. The general synchronization equation is:

$$t_{ref} + \delta \le t_{actual} \le t_{ref} + \varepsilon,$$

where δ is the minimum acceptable delay and ε is the maximum tolerable delay. The use of the minimum and maximum delay intervals is illustrated in figure 8.

Before considering the structure of synchronization arcs in detail, it is interesting to note that the concept of the synchronization arc maps well onto the threading concept on multiprocessor systems. Default synchronization arcs correspond to *fork* and *join* operations. Bounded (explicit) arcs can correspond to forks and joins that are either synchronous or asynchronous. Alternatively, the use of synchronization arcs can be mapped to the use of remote procedure call invocations in a loosely-couple multi-processor environment.

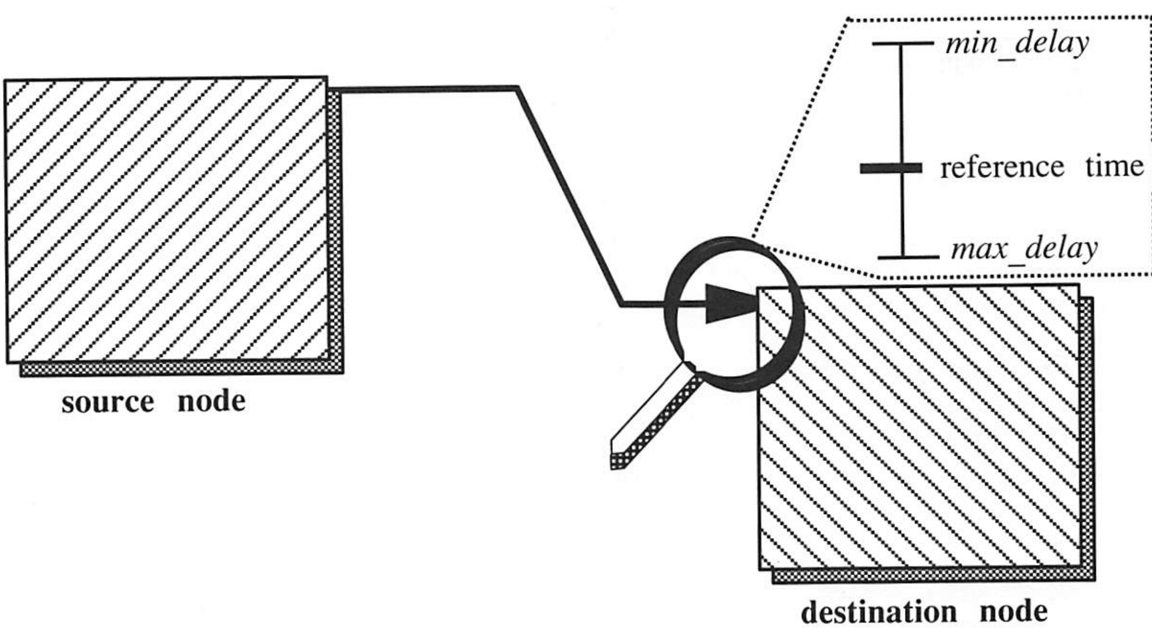


Figure 8: Synchronization delay parameters.

5.3.2. Specifying synchronization arcs.

Strictly speaking, synchronization in formation within a CMIF document is described via an *synchronization arc* attribute list. Given the importance of the synchronization arc attribute, however, we treat it separately in this section. The general structure of a synchronization arc is given in figure 9. Synchronization arcs are placed between two nodes, with the source being the controlling node. All nodes have an implied synchronization arc with the root node. Each of the fields consists of one or more attribute values that define how two nodes act relative to each other. The definition of each field is:

- *Type:* each type field has two components: an indication whether this synchronization arc concerns the beginning or the end of the event block being synchronized and an indication whether the synchronization is a ''must'' type or a ''may'' type. The meaning of begin/end is obvious. The meaning of May/Must is as follows: *May* synchronization is an indication to the implementation environment that the requested type of synchronization is desirable not but essential. This may be the case for the placement of label text at the beginning of our News example; if the label is a little late, then there is no reason for panic. *Must* synchronization is a stricter form. It tells the implementation environment that it (the environment) should do all it can to implement the requested type of synchronization, even at the expense of overall system performance. Considering the example once again, it is strictly required that each of the blocks associated with a story are presented together; the implementation environment has no freedom to delay sending one set of data if the request cannot be immediately honored.
- *Source, destination:* the source field specifies a relative path name in the tree (by using named nodes) for the controlling reference of an arc. The destination field specifies a relative path name in the tree (by using named nodes) for the target reference of an arc. The empty name specifies the current node itself.
- *Offset:* this field allows the synchronization to be defined relative to an integral positive offset from the start of the controlling node. Offsets may be expressed in terms of media-dependent units (such as seconds, frames, bytes, etc.).
- *Min-delay, Max-delay:* these are the minimum acceptable and maximum tolerable delay, as specified in section 5.3.1. These allow a document to compensate for different implementation environments.

CMIF allows absolute references to be defined by specifying an arc from the root with a minimum and maximum delay time of zero. Relative references are achieved by setting the source of the arc to be a predecessor node. Coincidental scheduling is possible by setting the source to be a peer node.

5.3.3. Potential synchronization conflicts.

There are three general synchronization conflicts that can arise in processing a multimedia document. First, an unreasonable synchronization constraint may have been defined (directly or indirectly) by a user. Second, device characteristics may limit the ability of a particular

type	source	offset	destination	min_delay	max_delay

Figure 9: Synchronization arc (in tabular form).

environment to support a given document. Third, in navigating through a document, a reader/viewer/listener may want to fast-forward (or fast-reverse) to a document section that contains a number of relative synchronization constraints for which the source or destination are not active. In general, each of these conflicts fall outside the scope of a document definition; they will be more appropriately handled by either user interface tools or document construction tools. Even though the document structure simply acts as a messenger in delivering documents for later processing, there are still several aspects of the document definition that can aid in detecting (if not correcting) the three conflict situation.

In the first case, an ''incorrect'' specification may have been made of timing relationships between two events. In the News example, the writer of the captions block may have a constraint that text must be displayed long enough to be readable by a user who is also interested in looking at the graphic material. In this case, a content-based checking tool may be required to signal conflicts. The document structure can assist in this task by providing a separation of the attributes that describe a data block from the block itself. Obviously, if the attribute information does not include timing information, there is little the document structure can do to help.

The second case is similar to the first, except that here the problem lies with the presentation of information rather than the contents of the information being presented. Here the solution is more straightforward. A local-constraint tool should be able to flag the conflict by studying information in the synchronization arcs. The implementation environment can then make a decision on whether or not to support the entire block or to perform a cut-off function or perhaps a ''stretch'' function on other data in the system. Once again, CMIF plays a role in signalling problems, allowing other mechanisms to provide solutions.

The third problem concerns itself with the implementation of the logical structure of the document. Because an internal tree is used to describe the data, the parents of a synchronization node can be traced until the common ancestor containing the source and destination of the arc is found. We support the general notion within relative arcs that the source of the arc must execute in order for a synchronization condition to be true; if this is not the case, all incoming synchronization arcs are considered to be invalid.

5.3.4. A synchronization example.

In order to bring together several of the concepts discussed above, consider the (contrived) fragment out of our newscast on stolen paintings presented in figure 10: In this example, each of the five synchronization channels presents a part of a complete story: the audio channel carries the announcer's voice with information (in Dutch) on the robbery; the video channel initially shows the announcer, then gives a view of the scene of the crime; the graphic window shows views of three of the missing paintings; the captioned text presents a translated version of the announcer's text; finally, the label field identifies the general scene with an occasional title.

In our example, the graphic channel is synchronized with the start of the audio portion of the report. Within the graphic channel, each illustration is sequentially synchronized. There is implied synchronization between the first and the second illustrations of the paintings and explicit synchronization among the second and third.

The captioned text is start-synchronized with the video portion of the display. (It is not synchronized at all with the audio; this allows one story to be presented for local consumption and another for global presentation.) A synchronization arc is drawn from the end of the second caption block to the start of the second graphic; this illustrates the use of an offset within an arc. At the end of the fourth caption block, an arc is drawn to the video portion to indicate that a new video sequence may not start until the caption text is over. This may require a freeze-frame video operation to support the synchronization.

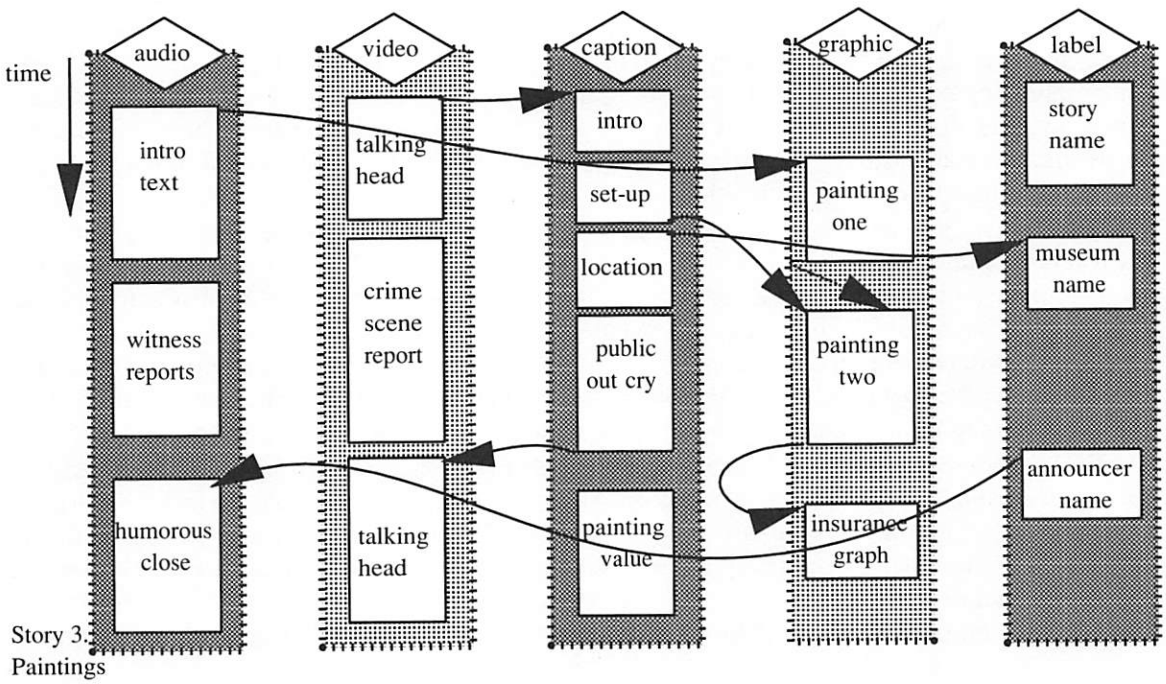


Figure 10: News report fragment structure.

Finally, the label channel occasionally displays generic titles that are linked to other portions of the display.

6. Research Directions

The sections above have provided an introduction to the CMIF structure that has glossed over a number of interesting problems that must be dealt with in a general multimedia environment. We will briefly review these problems in this section.

One of the first issues encountered in the specification of a transportable document is the resolution used in the definition of elements such as delay times, sound coordinates, sampling frequencies, etc. This is a particular aspect of a much wider problem: the specification of system-independent attributes for a document. While the temptation is great to minimize the number of attributes provided with either a data descriptor or an event descriptor, it is typically better to include as many attributes as is possible. This can be efficient for an application program (such as a constraint filtering tool) because such a tool will be given as much information as is possible upon which to make performance-related decisions. It is true that the support for a large number of attributes can make documents appear to be structurally heavy. This may give the perception that layers of attributes make a document large (and thus inefficient) as well as difficult to interpret. We do not believe this to be true for two reasons. First, by selecting appropriate attributes, much of the work associated with manipulating a document can be based on relatively small clusters of data (the attributes) rather than the often massive amounts of media-based data itself. Second, although we have created CMIF documents to be human-readable, our expectation is that the documents themselves will be created and viewed using appropriate user interface tools. The

use of many attributes can also serve to make a document more understandable even without a sophisticated viewing tool; unlike text strings, for example, most people find it difficult to meaningfully interpret bitmaps or audio sequences based on uninterpreted data encodings. (One could almost consider attributes to be an efficient document documentation technique.) In all cases, however, the difficult part of attribute support remains the selection of appropriate attributes to cover the needs of a wide range of media and implementation environments.

Given the selection of an appropriate attribute set, an intriguing question is the degree to which the attributes can be used to manipulate documents in a manner that is totally independent of the data itself. For example, if the attributes contain search key information, then many time consuming activities relating to finding detailed information in large multimedia database may be simplified. Another example is the use of attributes to include content-based links to other data structures. In both of these cases, the attribute list can be used as an interface between the raw data contained in a database and the encoded information manipulated by an application. Note that while the organization of a particular database is not necessarily related to the structure of a document using that data, an investigation into the ways in which manipulation of information can be made more efficient requires substantial coordination between data/event descriptors and the data management system.

One problem that is related to the two issues above is management of the location of data in a transportable document. While it may occasionally be necessary to move massive amounts of information from one computer to another (especially in situations where the computers are not directly interconnected), we also feel that the use of both distributed databases and distributed operating systems support is vital to the efficient implementation of multimedia systems. As with common documents (i.e., computer files), the value of document sharing and multiple access to information is vital to the effective sharing of information.

Each of these areas are receiving attention at CWI. In particular, we are investigating the use of the Amoeba distributed operating system [Mullender90] as a base for a distributed multimedia system, with integrated support for a distributed database mechanism to manage document storage across the multimedia environment. The selection of appropriate attributes and the efficient manipulation of document descriptions rather than document data is of primary concern to use. The over-all goals of this project are described in [Bulterman90].

7. Summary

The key to our approach of transportable document structure is in differentiating those aspects of a document that should be available in all environments from those that apply only to specific environments. We do not dwell on storage structure or on methods of encoding/compressing data for sharing in a heterogeneous environments. For us, a much more interesting problem is defining a document that can be used to control the processing of information rather than being a slave to that information itself. In doing so, it was important to classify the notion of time and the manner in which events took place relative to each other. It was also important, with an eye towards future research, to be able to build an abstraction for a data block that could then be manipulated separately from the (large) volume of data that such an abstraction represents.

8. Acknowledgments

The general ideas presented in this paper have benefited from discussions with many people at CWI and with several of our academic and industrial contacts. Dik T. Winter of CWI contributed substantially to a refinement of the presentation on synchronization issues. Lynda Hardman of OWL, Ltd provided a detailed and constructive review of the final manuscript. The Einstein illustration used in figure 4 was taken from a public-domain GIF file, and the iris illustration in the same figure was obtained from an image database provided by Silicon Graphics, Inc.

9. References

[Bulterman90] *Bulterman, D.C.A.*, "The CWI van Gogh Multimedia Research Project: Goals and Objectives," CWI Report CST-90.1004, 1990.

[Dean90] *Dean, Mascio, Ow, Sudar, and Mullikin,* "An Image Cytometry Data File Standard," Lawerence Livermore National Laboratory report, 1990.

[Frame89] *Frame Technology Corp.,* "The Frame MML Reference Manual," Documentation accompanying the FrameMaker 2.0 Publishing Software, Frame Technology Corp., San Jose CA, October 1989.

[Halasz90] *Halasz and Schwartz,* "The Dexter Hypertext Reference Model," NIST Hypertext Standardization Workshop, Gaithersburg MD, 1990.

[Hoffert91] *Hoffert and Gretsch,* "The Digital News System at EDUCOM," Communications of the ACM, Vol. 34, No. 4 (April 1991).

[Hodges89] *Hodges, Sasnett, and Ackerman,* "A Construction Set for Multimedia Applications," IEEE Software, Vol. 6, No. 1 (January 1989).

[Kipp90] *Kipp, Newcomb, and Newcomb,* "HyTime Review: Standard for Hypermedia/Time-Based Document Interchange," Submitted to Communications of the ACM for publication, 1990.

[Mullender90] *Mullender, van Rossum, Tanenbaum, van Renesse and van Staveren,* "Amoeba: A Distributed Operating System for the 1990s," IEEE Computer Magazine, Vol. 23, No. 5 (May 1990).

[Rossum91] *van Rossum, van Liere, Winter and Bulterman,* "The CWI Multimedia Interchange Format (CMIF) Reference Report," CWI Report VCST-91.502, 1991.

[Yankelovich89] *Yankelovich and Kahn,* "A Hypermedia Bibliography," Brown University Institute for Research in Information and Scholarship (IRIS) report, 1989.

[Thomas85] *Thomas, Forsdick, Crowley, Schaaf, Tomlinson, Travers, Robertson,* "Diamond: A Multimedia Message System Build on a Distributed Architecture," IEEE Computer, December 1985.

Biographical Information:

Dick Bulterman the project leader of the interdisciplinary research theme on Multimedia at CWI in Amsterdam. In addition, he has been the head of the Computer Systems and Telematics department at CWI since 1988 and has been an associate professor of computer science at the University of Utrecht since 1990. Prior to joining CWI, he was an assistant professor of engineering at Brown University. Dr. Bulterman received his Ph.D. in computer science from Brown in 1981. His research interest is the relationship between distributed operating systems and computer architecture.

Guido van Rossum is a research staff member of the multimedia project at CWI. Prior to this project, he developed the Python language for rapid prototype of interactive software, and he worked on the Amoeba distributed operating system and the ABC programming language. Mr. van Rossum received a degree in computer science from the University of Amsterdam. His research interests include rapid prototyping software, distributed operating systems and user interface software.

Robert van Liere is a member of the Interactive Systems department at CWI; he has been associated with the CWI/Multimedia project since that project's start in 1990. In addition to his work on multimedia systems, Mr. van Liere is the leader of CWI's scientific visualization project. Prior to joining CWI, Mr. van Liere, who studied at the Technological University of Delft, work at TNO, a Dutch government laboratory for applied science. His research interests include distributed systems support for computer graphics and user interface systems for scientific visualization.

Parsing Movies in Context

Thomas G. Aguierre Smith
Natalio C. Pincever

Interactive Cinema Group, The Media Lab
Massachusetts Institute of Technology

thomas@media-lab.media.mit.edu
Che@media-lab.media.mit.edu

Abstract

Traditional approaches in Multimedia systems force the user to segment video material into simple descriptions, which usually include endpoints and a brief text description in the form of keywords. We propose to segment *contextual information* into chunks rather than segmenting contiguous frames. The computer can help us organize sets of descriptions that are related to the recorded moving image: it can help us remember what we have shot. Such descriptions can overlap, be contained in, and even encompass a multitude of other descriptions. Parsing moving image sequences is reduced to simply parsing the contextual information which forms the description. Our approach, *Stratification*, also has important ramifications in terms of an elastic representation of moving images. Ambient sound can provide us with important contextual clues as to what is going on within the frames. Using audio to find patterns of content is an important step towards the eventual automatization of the logging process.

1 Approaches for Production: Segmentation versus Stratification

Segmentation forces the user to break down the raw footage into segments denoted by the begin and end point and assign some sort of textual description to them. In a video database application, these textual descriptions are searched and the associated video is retrieved. In our research we have found some problems with this approach.

On a multimedia system in which it is possible to access individual frames randomly, each frame needs to be described independently. The user has to represent the content of each chunk: the part is forsaken for the whole. In terms of granularity, the chunk of video that a database application can retrieve for a user is predetermined during the logging process. The computer representation of the video footage only encompasses a begin and end point and a text description. Sub samples or finer grained search criteria / representation have to be made independently. In terms of elasticity, if you have a segment which contains 30 frames; the application will retrieve all 30 frames when queried. It has no representation of any set of 10 or 20 frames which form a subset of the base frames. One would get 30 frames and then select a sub sample of these frames, describe them independently so that they can be called up later to satisfy a more finely grained search criteria.

Furthermore while segmentation is necessary when editing a video, it imposes a specific intentionality of the person who is placing the material in a structured context. Although this is desirable for a linear editing system, it can seriously impede a group of people who need to use and have access to the same video resources from an archive over a network. If the video material is segmented from the start, how then can the descriptive structures support other users who may need to access the same material for a different purpose?

Our solution is to segment contextual information into chunks rather than segmenting contiguous frames. This new context based approach is called *Stratification* (Aguierre Smith 1991). Begin and end frames are used to segment *contextual descriptions* for a *contiguous set of frames*. These

descriptions are called *strata*: a shot begins and ends; a character enters and sits down; the camera zooms in. Each represents a *stratum*. Any frame can have a variable number of strata associated with it or with a part of it (pixel). The content for *any* set of frames can be *derived* by examining the union of all the contextual descriptions that are associated with it. Before we can begin to think about parsing, we need a good representation of moving image content. Stratification is a way to structure video content information that will allow us the greatest latitude in terms of parsing. Stratification is a descriptive methodology which generates rich multi-layered descriptions that can be parsed by other applications.

1.1 The Management of Multimedia Resources

The movie maker knows a lot about the images as she is recording them. Knowledge about the content of the moving image is at its maximum while being recorded and drops to a minimum while being viewed. The computer can help us organize sets of descriptions that are related to the recorded moving image: it can help us remember what we have shot. Successive shots share a set of descriptive attributes that result from their proximity. These attributes are the *linear context* of the moving image. During recording, the linear context of the frames and the *environmental context* of the camera coincide. The environmental context is the "where," "who," "what," "when," "why," and "how" which relate to the scene: it's the physical space in which the recording takes place, giving it a unique identity. If you know enough about the environment in which you are shooting, you can derive a good description of the images that you have captured using stratification.

Clearly such descriptions can overlap, be contained in, and even encompass a multitude of other descriptions. Each descriptive attribute is an important contextual element: the union of several of these attributes produces the meaning or content for that piece of film. Moreover, each additional descriptive layer is automatically situated within the descriptive strata in which it already exits. In this way, rich descriptions of content can be built. On the other hand, segmentation, which is conventionally used in computer systems which allow the user to log video material, forces the user to break down raw footage into segments denoted by begin and end points: such a divide-and-conquer method forsakes the whole for the part. Coarser descriptions have to be included at this level of specification in order to describe one frame independently. If the unit represented by in and out points is as small as an individual frame, its description, in order to be independent, must encompass larger descriptive units. The granularity of description of sets of frames is directly related to the size of the image units being represented. In segmentation, the granularity of description is inversely proportional to the size of the image unit in question. The inverse relationship arises out of the need to describe each image unit independently.

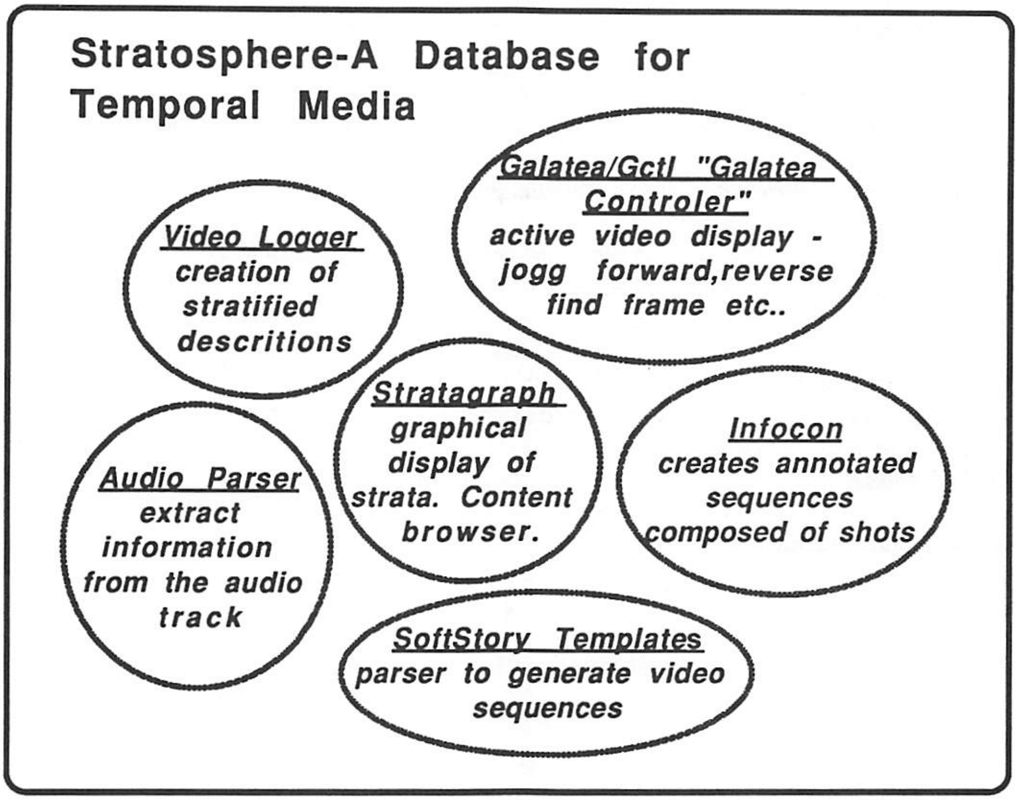
In addition to logging, film makers need tools which will enable them to take segments of raw footage and arrange them to create meaningful sequences. Editing is the process of selecting chunks of footage and sound and rearranging them into a temporal linear sequence (Davenport, Aguierre Smith, Pincever 1990). The edited linear sequence may bear no resemblance to the ambient reality that was in effect during recording. During the process of conventional editing, the original rushes become separated from the final motion picture. In order to create a motion picture with a new meaning which is distinct from the raw footage, a new physical object must be created - an edited print for film, or an edited master for video.

Editing on a digital-movie-database system will be radically different. In a full digital system, the link between source material and final movie *does not have to be broken.* The shot in the final version of the movie being made will be the same chunk of video data as the source material. At this point there are two names for the image unit which must coexist: one reflects the context of the source material; the other is an annotation that is related to a playback time for a personalized movie script.

1.2 Stratosphere

We have developed an application called **Stratosphere** which runs on a UNIX workstation under X/Motif using the Galatea video server (Applebaum 1990). Stratosphere (see Diagram 1) consists of a logging module, a stratagraph to display contextual description of frames through time, an icon-based video annotating system called Infocon and soft -story templates.

Diagram 1: Stratosphere



Stratosphere is an implementation of the Stratification method for representing moving image content. Stratification allows the user to keep track of contextual descriptions of video material and helps maintain the integrity of the contextual information during all phases of production.

In our UNIX video database environment, the distinction between *source* and final *edit* become so blurred that all video becomes a *resource* which can be used in different contexts (movies). Conceivably, one can edit the source material in the video database into a documentary film that will be played back on the computer. Moreover, these "edited-versions" can later be used by someone else to make another moving image production. The process of editing using Stratification becomes the process of creating context annotations, and storing them along with the initial descriptions made during recording. We are developing tools for video databases that will enable the user to semantically manipulate video **resources** which allows for alternative "readings/significations/edits" of the same material to co-exist on the system. Video resource can exist in many different contexts and in many personalized movie scripts.

When a user annotates a segment of video, s/he is creating a new stratum that is associated with a particular moving image in an edit. The old strata lines still exist in the database, but in editing a new meaning emerges in the re-juxtapositions of image units. This new meaning takes the form of an annotation and is displayed on the stratification graph (see figure 3). In a sense, the content of a series of frames is defined during logging. Yet the significance of those frames gets refined and built up through use. This information is valuable for individuals who want to use the same video resources over a network in order to communicate with each other.

2 The Stratosphere Video Database Modules

As mentioned earlier, Stratosphere is composed of separate modules which allow the user to log, manipulate, manage and view chunks of video material which have been described using the Stratification method. In this iteration of Stratosphere, we are using a MicroVaxII computer with a Parallax video card, UNIX /X Window System, Galatea video server, and Akai PG1000 digital matrix patchbay which manages a system of six video disc players.

2.1 The Logger

The Video logger is a Motif window made up of buttons, menus and field widgets which allow the user to select and describe video material. Logging is coordinated with browsing the video material displayed in the "gctl" (Galatea control device) window. Using the mouse and keyboard the user can fast-forward, review, search for a particular frame number. Gctl also has a video slider which enables the user to quickly scroll through video at various speeds in both forward and backward directions with only a click of the mouse. When the mouse is released, the video is stopped within one or two frames. Once the desired in-point end point is found for a descriptive attribute, the user presses the button and the current frame number is displayed. Next, a title and a set of key words are also entered for each stratum.

Describing video with keywords can be problematic. The choice of keywords is related the users intentions; they reflect the purposes and goals of a given multimedia project. We have address this problem by organizing keywords into classes. Keyword classes are sets of related words which can be created, stored and be reused for different projects. Each keyword class is stored as an ASCII text file. If desired, the user can edit the keyword class file with any UNIX text editor. When the user opens a class file, a button widget is created and displayed in the Logger window. Many different classes can be loaded during a logging session. When the user wants to associate a camera keyword with a strata line, s/he clicks on the camera class button and the all the keywords of this class are displayed in the keyword text widget. Double clicking on a particular keyword selects and enters it into the data file. Keywords classes can be combined to create customized keyword for sets. Key word classes speed data entry. For example, the "camera" keyword class which contains, wide shot, medium shot, pan left, pan right, zoom etc. is used over and over again.

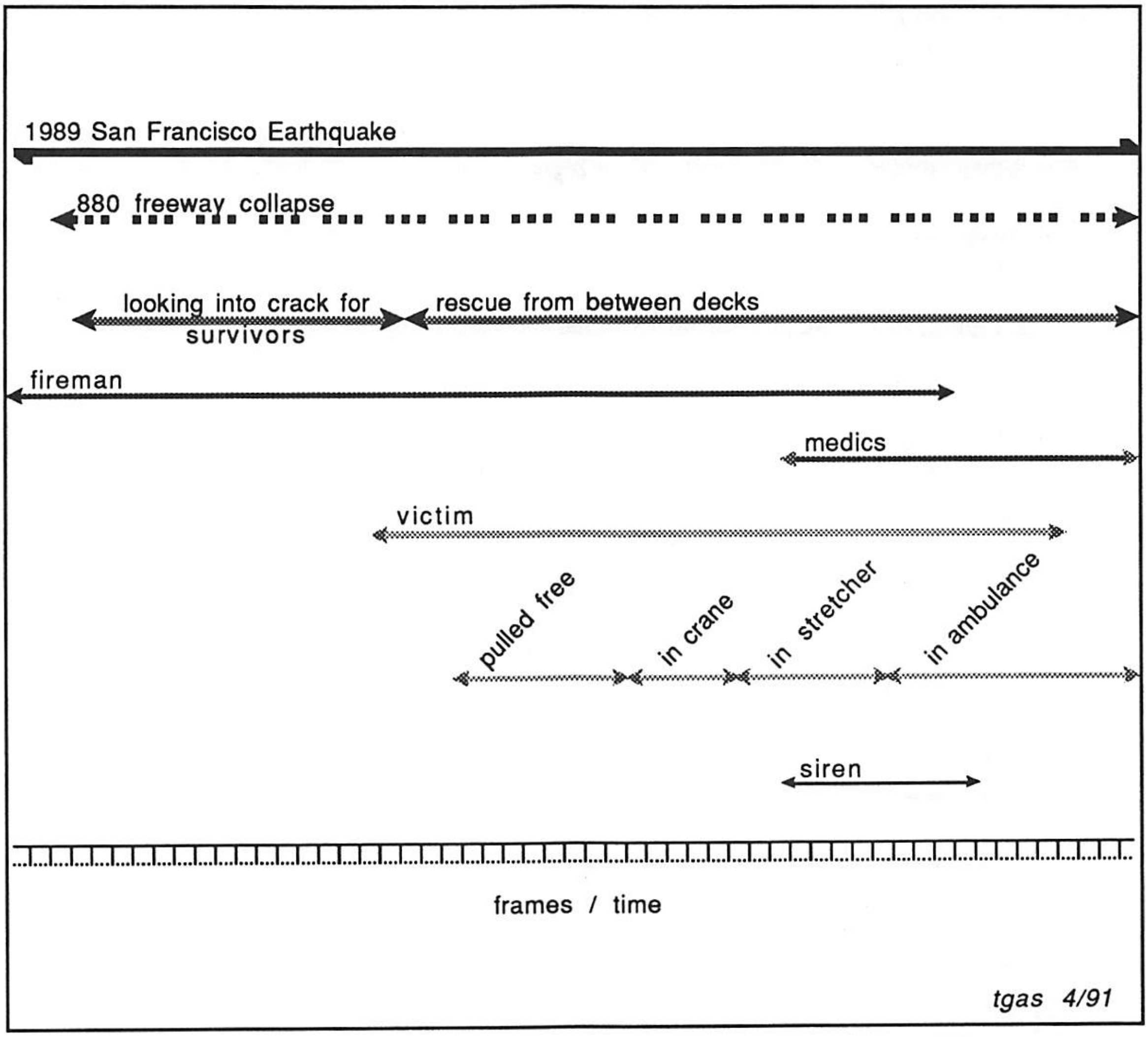
2.2 Stratagraph and Strata Data Display

The Stratagraph is an interactive graphical display of the data generated by the Logger. It is a visual representation of the occurrence of strata though time. An example of a Stratagraph is show in Figure 1, each descriptive stratum partially describes a set of frames; the union of all the strata is the content. The Stratagraph severs as a visual aid for logging video, since the user can log several descriptive attributes simultaneously.

During the logging process, each descriptive attribute is first anchored to the Stratagraph with an in-point and a strata line is generated for each frame that follows. At this moment, the user creates a title for the stratum and selects keywords in the Logger window. When the descriptive attribute is no longer in effect, the user clicks the strata line to turn it off.

We are investigating how text and color can enhance the readability of such graphs. Two different x-axis scales are used to display the stratagraph: the real time scale and the "scale of difference." For the real-time display, the scale of the x-axis is time code (frame numbers for laserdiscs). This scale is well suited for browsing through video material. A current frame line (not shown) passes through the stratification graph in sync to the video displayed on the "gctl." The scale of difference is a compressed graphical representation of content where only changes in descriptive attributes are displayed.

Figure 1: Rescue After the San Francisco Earthquake - Stratified Description



When the user "rubs" the stratagraph with the mouse, detailed contextual information for those frames are displayed in an associated Strata data window. A strata rub generates a report of all the strata that the frames are embedded in. The Strata data window is also used with the Infocon module.

2.3 Making Sequence Annotations: Infocon

During this phase of production, shots are delimited by in and out point and annotated so that they can be played independently. To create a new sequence the user inspects the Stratagraph for material with the desired content to select a shot, reviews the footage in GCTL and annotates these shots in the Infocon module. The Infocon (INFOrmation iCON) module consists of text widgets for annotations, buttons to grab frame numbers from Galatea, and video windows to display digitized in and out points. When Infocon retrieves the in- and out-frame number from Galatea, the associated frames are digitized and displayed in video windows on the Infocon module.

Although in and out point are important cues for visual continuity, they might not provide an adequate representation of the shot content. In addition to the in and out frame, a content frame is digitized and its associated frame number is also stored with the data recorded for the shot. The content frame is used to create an Infocon which serves as a visual representation of shot content. An Infocon is composed of a video window and a title bar which displays the annotation associated with the shot that it represents. Left-clicking in an Infocon, plays the shot in the GCTL module. Middle clicking on an Infocon displays a

pop-up menu which tells which volume name, in and out points, trim button, delete, button and a exit button to hide the pop-up. The in and out frame for a given shot is only displayed in the Infocon module while the Infocon for each shot is displayed in its own window. Infocons for each shot are browsed and arranged from left to right on the screen. Ordered groups of Infocons viewed as a sequence. Later, they can be reordered and played back. If needed, in and out points may be trimmed. When a satisfactory sequence is arranged, annotations, volume-name, in and out frame number are saved as a record for each shot. These records are then saved to an ASCII text file. These sequence files can then be loaded into the Stratagraph and the strata for the annotations are displayed with the original logging data. In this version of the Stratosphere application only annotations and their associated in and out and content frame numbers are stored. The digitized video images are generated on the fly, when a particular sequence file is loaded into Infocon.

In figures 2 and 3, we illustrate how a virtual annotated sequence is created. In figure 2, the user inspects the Stratagraph to select shot content. The in- and out-points are digitized by the Infocon module and a content Icon is created. In figure 3, these shots are arranged to create a virtual sequence.

Figure 2: Shot selection for "Last Survivor Sequence"

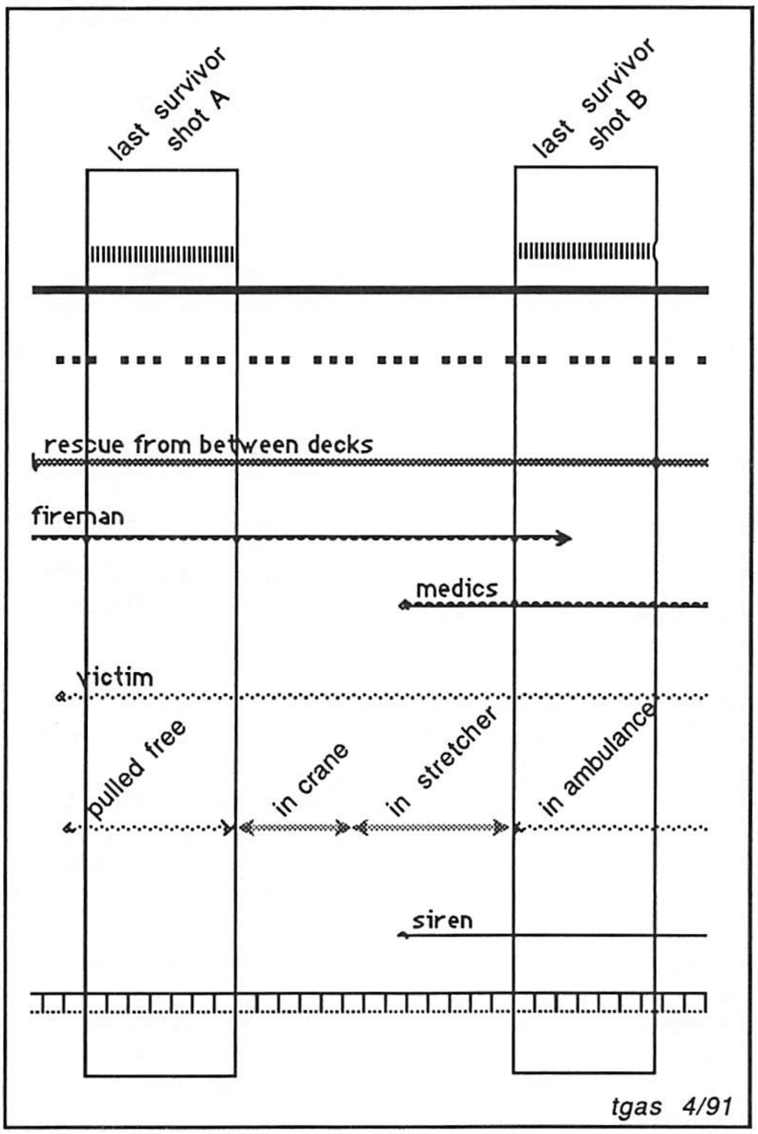
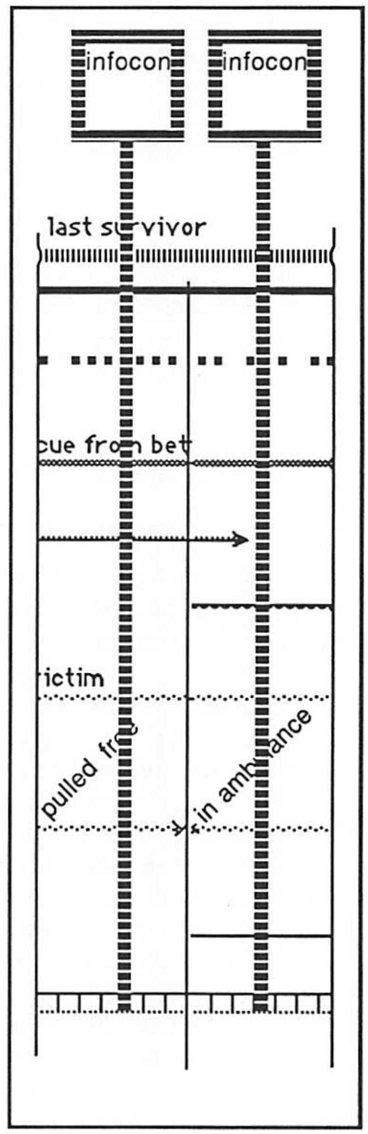


Figure 3: Display of Virtual Annotated Sequence - Last Survivor with Infocons



The bold-dashed line in Figure 3 is an icon-line. The description of the shot can be derived by examining the strata that this line passes through (Table 1)

Table 1: Content for Infocons in Figure 3

Infocon A: Last Survivor A	Infocon B: Last Survivor B
1989 San Francisco Earthquake	1989 San Francisco Earthquake
880 freeway collapse	880 freeway collapse
rescue from between decks	rescue from between decks
fireman	medic
victim	victim
pulled free	in ambulance
	siren

When viewed, the virtual annotated sequence appears to be made up of adjacent shots; but as we can see in figure 2 this does not necessarily have to be the case. The stratagraph is used to graphically represent both logging information and annotated virtual sequences.

In the scenario just described, the user manually selected the shot to be used in the "Last Survivor" sequence. Infocon can also generate icons on the fly for any set of strata provided that the data file is in the correct format. A file of annotations can be parsed using UNIX commands such as "grep," "sort," and "awk" and then sent to Infocon to generate icons.

2.4 Soft Story Templates

The ASCII data files created by the Logger and Infocon modules are represented on the stratagraph; they can also be "piped" through soft story templates to generate mini-video programs. An example of a soft story template is a news segment generator called ACE. ACE parses strata lines to find contiguous sets of frames which satisfy search and format requirements and then arranges them into sequences. ACE has a rule to construct a news sequence which consists of:

- a studio shot as an introduction;
- a predetermined number of action shots;
- a reporter in the field;
- a wrap up studio shot.

3 Audio Applications: Editing Assistance Through Cinematic Parsing

The shot, generated by the camera, is inherently associated with the circumstances and intentions which were in effect as it was being recorded. A similar case can be made for synchronous sound: as with the camera, the microphone/sound recorder mediates the environment in which it is situated. Ambient sound offers us clues about the physical and geographic aspects of place. When parsed, the original synchronous sound track for any movie may help us segment the video/audio stream into shots or help us determine significant action points. Audio tracks of home movies are essentially composed of the same elements as all movies: ambient sound and meaningful dialog. Yet it is in the ambient sound where we find the most interesting distinction. In narrative movies, this element plays a secondary role: most ambient noise is undesirable, and is eventually eliminated. In fact, many re-shoots are caused by unwanted changes in ambient noise, like a plane flying overhead. This is mainly true due to the fact that such noises are extremely difficult to edit. The only type of ambient noise generally used is what's called "room tone," which is used to create the mood of the space in which the action is supposed to occur.

We take another approach: ambient sound can provide us with important contextual clues as to what is going on in the frames. Ambient sound in raw footage has all its original components present, from cars going by to hand adjustments on the camera. Room tone becomes more than just ambiance: it becomes the specific attribute of the time and space in which the shooting is being done. Several audio techniques can be used to obtain the information needed to extract said attributes.

3.1 Techniques Used to Parse Audio Tracks

Statistical properties of the audio signal, like mean and mean-squared, can be computed for analysis purposes. They can tell us where changes in the power of the signal occur, which can clue to drastic changes in background noise levels. Using domain-restricted knowledge, such processing can give an indication of the type of changes happening: a rapid increase followed by a rapid decrease is usually a thump, caused by someone shutting a door, or dropping something heavy on the floor, or grandma falling off her chair.

We use several techniques to analyze the spectrum of the signal. Of interest are average spectrum and average amplitude (sum of mean-squared spectrum). These techniques give us information useful to determine changes in the power distribution and the spectral content of a signal. This techniques are used in the following way: we create a template with the spectral characteristics of a sample window, which is then compared to the following window. If there are considerable changes in the power distribution, then we assume that the spectral content of this second window is different, thus capturing changes in the *nature* of the signal, not changes in the amplitude. A combination of both time- and frequency domain analysis should suffice to capture enough changes in signal content to find the shot boundaries.

Using audio as an indication of patterns of content could prove quite useful. The audio track can provide information that otherwise would be extremely difficult to parse. Take, for example, a funny incident: the family is gathered in the living -room, and grandma fell off her chair. As stated earlier, there's no way of telling whether this was just a thump, or was it something more interesting. One way to do this is to use laughter. If it is possible to parse laughter, then we can tell that something funny happened, thus it is not desirable to remove it from the track.

Another interesting possibility is the development of templates, or "spectral signatures" of different sounds, to keep for later comparison. For example, assume that the system has stored the spectral content of a police car passing by. By checking through the soundtrack for occurrences of this spectral pattern, the system can tell that within a certain segment, there's a police car going by. This could lead, provided there was enough memory for storing patterns and enough computing power to perform the comparisons, to the complete automation of the logging process. This ties in into the stratified representation approach. Once a specific attribute is found, it becomes a part of the stratified description of the shot being examined. In the last example, a police car passing by was found on a particular shot. This could then be incorporated into the strata graph, as a shot description: "A car passes by." The converse is also true: the strata graph can provide information that, used with the "spectral signature" of the particular sound, can create new templates. For example, assume that we have a car on Pismo Beach. Parsing the audio track, we've found that the "car" template is present on this shot. We also know from the strata graph that this particular car is on the beach. We can then use the audio corresponding to that particular segment as "car on the beach" and save it for fine-grained parsing.

3.2 The Parser: Implementation

It is in the parser that the concepts and methods discussed above are implemented. The parser has two different modules: the digital signal processor, and the rules module. The digital signal processor does the actual parsing required. To achieve this, it performs several tasks, such as Fast Fourier Transforms (FFTs), Linear Predictive Coding (LPC), and statistical analysis. The rules module consists of a set of heuristics, which make certain "guesses" as to how to interpret the information extracted by the signal processing module. The parsing itself was done in different layers, each giving different information that could not be extracted from the others.

The first layer consists of time-domain analysis of the sampled waveform. The soundfile is divided into chunks, each being the equivalent of a frame of video. The number of samples representing a frames (1470) is obtained by dividing the sampling rate (44.1 KHz) by the number of frames in a second (30 for NTSC video). The amplitude mean is then obtained for each block. Using heuristics based on observation of raw footage, some preliminary conclusions are then drawn from the observation of these mean values.

This second layer consists of frequency domain analysis, after preforming a Short-Time-Fourier Transform on the soundfile. Analysis done at this level consist of standard signal processing techniques

described in an earlier section. These two analysis schemes detect changes in the energy distribution on all frequency bands, thus capturing changes in the *nature* of the signal, not changes in the amplitude. This second-level processing is only performed for cases that are ambiguous at the first level, meaning that a clear distinction cannot be made simply with amplitude changes.

Another interesting issue that comes up during this second layer of parsing is that of time duration of events, or granularity. How long does an event have to last in order to be considered a new unit, instead of a short occurrence within another? This is perhaps the most interesting feature of the parser: it takes the lowest average amplitude for a sample window as the background noise level. If this is only the background noise level, then no segment can have a lower amplitude; it must be the absolute minimum. Thus, if the overall signal level eventually comes back down to the saved minimum, then the background noise level has not changed.

3.4 The Parser: Applications

For example, let's assume that the mean increases sharply, to then decrease as sharply as it increased after a short period of time (a "spike"). The heuristic in this case would be that this was a thump, caused by a slamming door, an accidental tap on the camera microphone, or a loud knock. Another example would be a smooth increase followed by a smooth decrease. The heuristic for this case could yield several results: a passing car, an airplane flying overhead. None of these would qualify as a shot change, since they are transitory changes in background noise levels. There are other cases which could possibly be shot transitions, but the information available at this stage of processing would be insufficient. For example, a smooth increase in average amplitude. This could mean that a car drove into frame and stopped, that someone turned on a noisy device (like a fan or a computer), or it could also represent a smooth pan (which could mean a shot transition). In these cases, a second layer of processing would be required.

The following figure shows an actual segment of the audio track taken from a home movie. It is a simple amplitude vs. time representation.

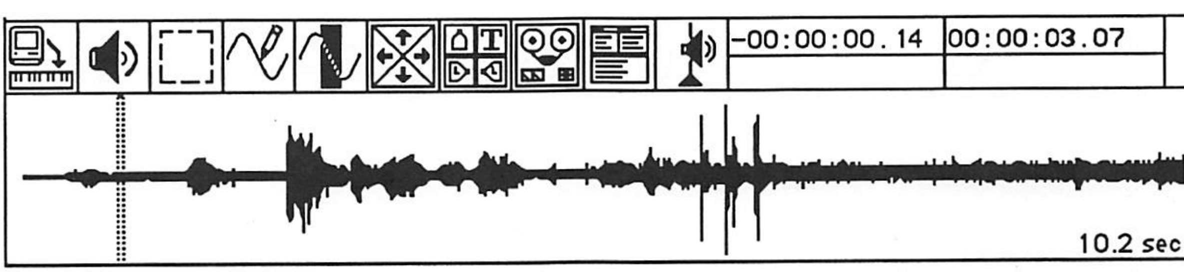
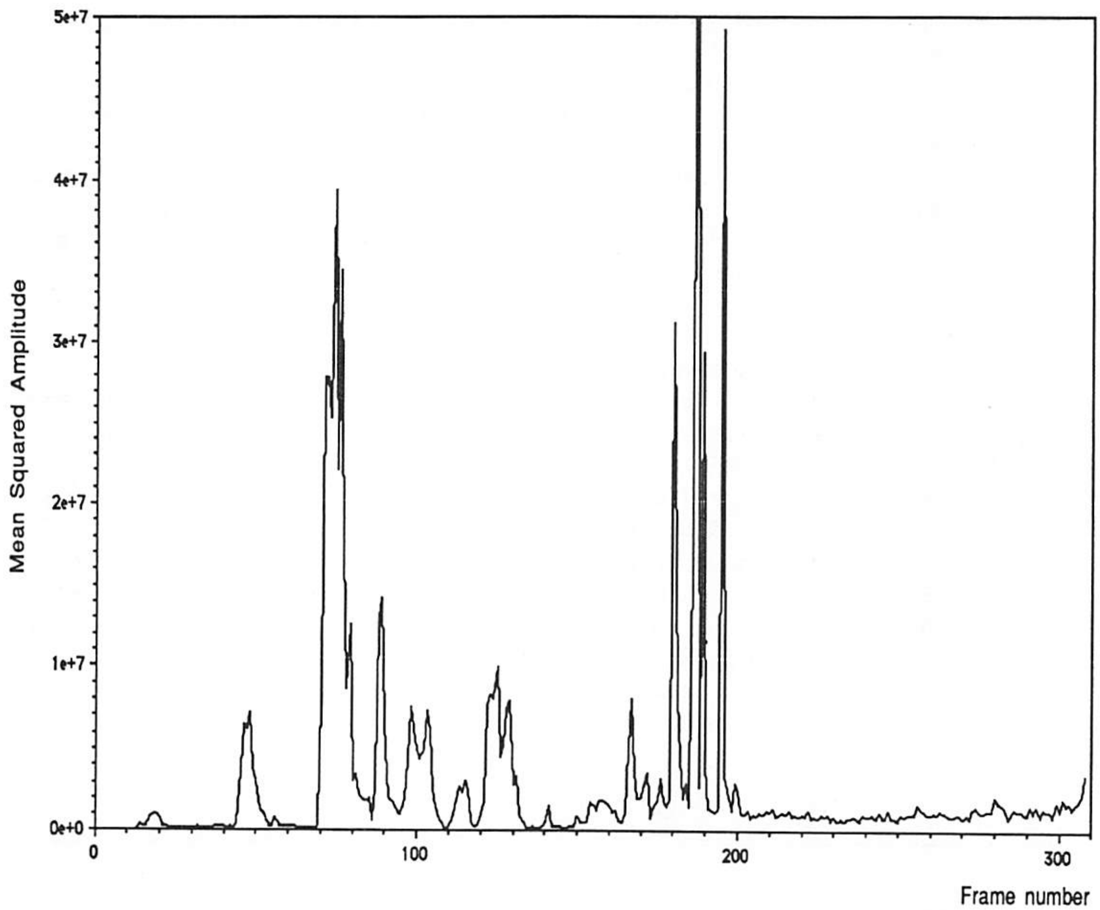


FIGURE 4: Soundfile representation- Amplitude vs. Time

The first four changes in amplitude are representations of human speech, while the four spikes towards the end represent hand adjustment after a shot change. The transition from one shot to the other is in between the two groups. The figure that follows shows how the parser interprets this audio segment.

Figure 5: Parser Interpretation Shot Transition Soundfile



This shows the first pass at the parser: the amplitude layer. Mean-squared amplitude is shown on the vertical axis, while the frame number is shown on the horizontal. The parser will eliminate the speech part, using the timing constraints as well as the fact that the amplitude eventually returns top it's original level. This will cause the parser to select the region between frames 160 and 170 as the one with a possible transition. Second level parsing will then show the change occurring at 165 frames, using average spectral amplitude differences.

4 Conclusions: Parsing Video In Context

Moving images, sound, and text are the base components of complex multimedia systems. We define these media as *resources* which can be used simultaneously in multiple contexts and applications. A context-based representation is robust enough to describe video resource and sound resources that can co-exist in many different contexts. Without knowledge about the context, descriptions become ambiguous. Another problem caused by such description is the lack of an elastic representation. Elasticity pertains to the use of multimedia data: how can various individuals who will have access to the same set of multimedia data, satisfy the information requirements of their own needs? For example, some individuals will want to use all 5 minutes of a videotaped interview while others will only want the last 20 seconds of the interview. Others still will only want one frame of the interview to use as a picture in a newsletter.

The Stratification methodology breaks down the environmental factors into distinct descriptive threads called *stratum*. When these threads are layered on top of one another they produce a descriptive

strata from which inferences concerning the content of each frame can be derived. Applications are being built that apply this methodology to logging and editing.

Contextual representations of audio are based on the ambient sound present in raw footage. This ambient sound becomes the specific attribute of the set of frames being examined. Through the use of heuristics, statistical analysis, and signal processing, changes in ambient sound can be parsed. Further application of these ideas can lead to the automation of the logging process through the use of templates, provided enough computing power was available.

Elasticity in description and searching is especially important when authoring a multimedia system. Our research points to new ways in which the user can interact with multimedia systems by varying the length of various chunks of video on demand. The representation gives the user maximum flexibility in manipulating multimedia resources. The value of each type of description, both sound and image, is enhanced by the other. This will lead to an integrated representation of both image and sound descriptions.

Acknowledgements

The authors would like to thank Joshua Holden, Kristine Ma, Lee Morgenroth, Carlos Reategui, and Derrick Yim for their assistance on implementation details; Hiroshi Ikeda, research affiliate from Asahi Broadcasting Corporation, for his work on Infocon; Mike Hawley, for DSP ideas and useful comments; and Glorianna Davenport, for comments, suppport, and guidance.

Bibliography

Aguierre Smith, Thomas G. "Stratification: Toward a Computer Representation of the Moving Image," working paper, Interactive Cinema Group, MIT Media Lab, January 1991.

Applebaum, Daniel. "The Galatea Network Video Device Control System, MIT, 1990.

Davenport, G. and W. Mackay. "Virtual video editing in interactive Multimedia Applications," Communications of the ACM, July 1989.

Davenport, Glorianna, Thomas G. Aguierre Smith and Natalio Pincever , "Cinematic Primitives for Multimedia: Toward a more profound intersection of cinematic knowledge and computer science representation." to appear in special issue of IEEE Computer Graphics and Applications on Multimedia, to be published Summer 1991.

Ellis, Dan. *Some software resources for digital audio in UNIX*, Music and Cognition Group technical document, MIT Media Lab, April 1991.

Handel,, Stephen. *Listening*, MIT Press, Cambridge, 1989.

Ludwig, L. and D. Dunn. "Laboratory for Emulation and Study of Integrated and coordinated media communication." Proc. ACM SIGComm, 1987.

Pincever, Natalio. "If You Could See What I Hear: Editing Assistance Through Cinematic Parsing." Master's Thesis, MIT Media Lab, June 1991.

Purcell, P. and D. Applebaum. "Light table: an Interface to Visual Information Systems," in *Electronic Design Studio*, MIT Press, Cambridge, June 1990.

Sasnett, Russell Mayo. "Reconfigurable Video." Master's Thesis, MIT Media Lab, February 1986.

Thomas G. Aguierre Smith is a graduate student in the Interactive Cinema Group at the MIT Media Lab. Research interests include multi-user networked video databases, problems related to on-line video archiving, video indexing systems, and how video can be used as a research tool. His current research involves developing a computer representation of moving image content called "Stratification." Smith received a BA degree in Social Science with a concentration in Anthropology at the University of California at Berkeley.

Natalio C. Pincever is working on his MS degree in Media Arts and Sciences at the MIT Media Lab. His current research is focused on cinematic parsing of audio tracks for the Interactive Cinema Group. His research interests include digital audio processing, desktop audio, advanced human interfaces, multimedia communication and hypermedia. Pincever holds a BS degree in Electrical Engineering from the University of Florida. He is a member of the IEEE Computer Society.

Distributed Multimedia: How Can the Necessary Data Rates be Supported?

Michael Pasieka (msp@andrew.cmu.edu)
Paul Crumley (pgc@andrew.cmu.edu)
Ann Marks (annm@andrew.cmu.edu)
Ann Infortuna (ai0d@andrew.cmu.edu)
Information Technology Center
Carnegie Mellon University

Abstract

At the Information Technology Center at Carnegie Mellon University[1], we have been developing a system in which it is possible to deliver data to a presentation machine from a remote machine across a public network at a sustained high rate. We have called this system a *Continuous Time Media System* (CTMS). We have found that the UNIX[2] model used for transfer of data between two devices, the network transport protocols, and the ability of adapters to transfer data between themselves are insufficient to do this. We have made modifications to each of these system in order to create a prototype system that we can measure to help define a CTMS protocol. We present the results of this work in this paper.

1. Introduction

The word *multimedia* has been widely used and misused in referring to a computer's capability to present and manipulate non-textual data. A *multimedia system* often implies a system that can manipulate only text and graphic. Such a system may be incapable of displaying both types of information simultaneously. There may be no method for linking the text with the graphic and no means of accessing the information from a remote machine, but the system may still be called multimedia. An ideal multimedia system might include text, graphic and tabular data (including spreadsheets), vector drawings, animations, still images, full motion full color video, Compact Disc quality audio and whole interactive programs (such as a full functional calculator or piano keyboard), all of which could be included within one document. The system might also allow for arbitrarily complex linking of data, and distributed data. Although, few systems exist that approach this ideal, when we discuss multimedia systems we want to consider this whole system.

One key limitation of systems that include full motion, full color video (of NTSC quality) or Compact Disc quality audio is that they are stand alone systems or systems that use private networks to transport the multimedia data. Until now, no one has been able to bring to the market or the lab a real time, public local area networked system for such high data rate media.

We define a *Continuous Time Media System* (CTMS) as a system in which the data to be presented must be received at a reliable, continuous high rate. This presents concurrently the problems of high volume data transport and real time response to the transmission and reception of the data. For example, with Compact Disc audio, the transfer rate is 176.4KBytes/sec (44.1K samples, 16 bits per sample, 2 channels). The source machine must read a disc and redirect the data flow onto the local area network. The destination machine must then receive the data from the network and redirect the flow to the subsystem

---

[1]This work was performed as a joint project of Carnegie Mellon University and the IBM Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies of Carnegie Mellon University or the IBM Corporation.

[2]*UNIX* is a registered trade mark of AT&T. *RT/PC* and *Token Ring* are registered trade marks of IBM. *AFS* is a trademark of Transarc Corporation.

that is converting the digital data to audio in a such a way that no discernible glitches are heard. This presentation is not simple, even in a stand alone system.

Our group works within the following operational environment. The base operating system is Academic Operating System (AOS) 4.3, a variation of BSD 4.3 UNIX that runs on the IBM RT/PC machines. The system is also an Andrew File System (AFS) client [Morris86]. The local area network is a 4Mbit Token Ring with 70 machines of which several are file servers running AFS. Note that this paper only addresses the problems associated with data transport of CTMS over a Token Ring network in which the source and destination machines are on the same local network (i.e. source and destination are not separated by a router). Although we could have chosen other operating systems and other machines, availability and established expertise were a major factor in these decisions.

Our goal was to support the data rates needed by CTMS over a 4Mbit Token Ring local area network while the ring was in use for other data transport. We ran several tests of the current UNIX system, described above. The initial test was to transport 16KBytes/sec of audio data (8K samples/sec, 12 bit/sample). This worked extremely well within the current UNIX model. We then tested the use of 150KBytes/sec to simulate compressed video or Compact Disc quality audio. This test of data transport failed completely. In this paper, we present our proposed changes, the measurement tools we used to measure the modified system, as well as some of our thoughts about the actual measurement data. With our proposed changes, we created a prototype for successfully transporting CTMS data over a 4Mbit Token Ring local area network, which was loaded with other data.

2. The UNIX Model of Device to Device Transfer and CTMS

In the current UNIX model of data transfer, the only method of transfer between two devices is to create a user level process that reads the data from one device and writes the data to a second device. This leads to having too many data copy operations, which then leads to the CPU's inability to maintain the necessary data rate.

Referring to Figure 2-1, if a user level process reads data from a device, at least two data copies are performed. The first is normally a Direct Memory Access (DMA) transfer between the device and kernel

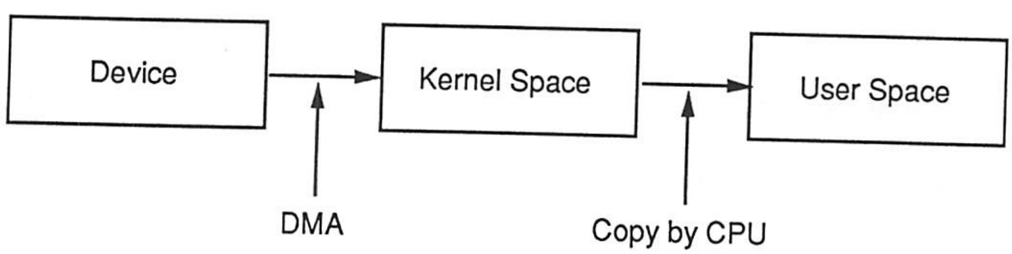


Figure 2-1: Data Copies From Device to User

space[3]. If we can ignore the loading of the system bus while the transfer is in progress, there is no loading

[3]There is at least one device currently available, the Audio Capture and Playback Adapter produced by IBM, which does not use DMA. In fact, the interface is a byte wide interface. I must assume that the designers of the adapter expected that the audio data would be compressed in software on the adapter before being transferred to the host system. Of course, this requires the device driver builder to also program the DSP (in this case a TI32025) to do this compression. It should be noted that there also exist adapters that use on-card memory mapped into system memory as the method for data transfer.

of the CPU to do this particular transfer. If the CPU is executing a memory intensive computation at the time, the arbitration between the DMA and the CPU access will degrade the execution speed of both. This DMA transfer always occurs into kernel space.

The second copy is performed by the CPU to transfer the data between kernel space and user space. Unfortunately, the implementation of most device drivers includes a third copy of the data. This third copy is done by the CPU. Referring to Figure 2-2, device drivers normally use fixed DMA buffers in kernel system memory. This forces the device driver to copy the data out of the fixed DMA buffers into a linked list of kernel buffers called *mbufs*.

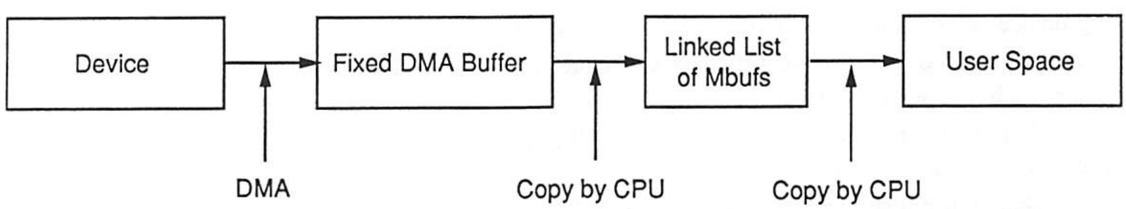


Figure 2-2: Expanded Data Copies From Device to User

The UNIX model uses *mbufs* as a pool of buffers to transfer data between the various layers of protocols. The device driver allocates *mbufs* and copies the packet into these *mbufs*. It should be noted that the allocation of a *mbuf* can be delayed an arbitrarily long time if the pool is exhausted at the time of the request. To transfer the data back to the second device requires three additional data copies. Two of these copies are performed by the CPU and one is performed via DMA.

In total, the number of copies performed to effect the transfer of data between two devices can be as many as six and as few as four. The difference of two copies can be accounted for by the devices' DMA capabilities. There will always be four copies made by the CPU. At a minimum, two of these copies are unnecessary.

We propose a change to the UNIX model of data transfer between two devices. Specifically, direct driver to driver data transfers should be done. This completely eliminates two of the data copies. This change requires that the source device be given a function which when executed will effect the transfer of data between the two devices. In the case where the network is the source of data, an additional function is needed. This function needs an argument of a linked list of *mbufs* that is a newly received packet. This function returns true when the packet should be directly transferred to the device. Handles to these two function calls can be transferred by a user process between the two devices by using newly created *ioctl* calls. We implemented this change of direct driver to driver data transfers in the prototype system for CTMS.

There is one further change that can be made if the model of UNIX data transfers is extended to include transfers by pointer manipulation rather than by data copy. Given that both devices are capable of DMA, all CPU data copies can be eliminated by transferring pointers to DMA buffers between the two devices. If only one of the two devices is capable of DMA, then only one copy can be eliminated.

3. Local Area Network Protocols and CTMS

The currently available standards of Transmission Control Protocol (TCP) and Internet Protocol (IP) as local area network protocols for data transport are insufficient for the data transport of CTMS. Several guarantees are needed from the protocol used to transport the data. These include:

- Bandwidth across the network
- Delivery of a packet within preset time bounds
- Preservation of packet sequence

These guarantees are necessary so that both the amount of buffer space used and the amount of processing time for any single packet are bounded. Of the three guarantees, TCP/IP only provides for one: the preservation of packet sequence. These protocols can guarantee the preservation only by creating more network traffic in the form of acknowledgments and requests for retransmission of lost packets. The model of the network that was used when these protocols were developed included the idea that the physical network was unreliable. Times have changed.

If a Token Ring network is used, it is possible for the transmitter at a hardware interrupt level to know if the packet was successfully received at the destination[4]. Given this, the levels of software previously used to guarantee packet delivery can be pushed down into the interrupt handler. If the Token Ring device driver is also constrained to send one packet completely before attempting to start sending another, a guarantee of preservation of packet sequence can be accomplished by giving the device driver the packets in the order required.

Additionally, the TCP/IP protocols make the assumption that the network can be dynamically reconfigured. As a consequence of this assumption, IP requests the Token Ring header be recomputed for each packet transmitted. In our case, the transmitter and receiver are always on the same local area network[5]. If we use IP, this would add an additional delay and load on the CPU for no reason, since we know that the route never changes.

Given the requirements that were not met, and other limitations, we decided that TCP/IP would be insufficient as network protocols. We propose that a new protocol be created, CTMS Protocol (CTMSP), and added to the same layer as ARP and IP. This protocol is specifically designed for and limited to the assist of data transfers between the network and other devices. The protocol assumes a static point-to-point connection between two machines. The implementation changes required to add CTMSP to the Token Ring device driver include:

- Adding the ability to specify the priority of the packet sent over the Token Ring. CTMSP uses a Token Ring priority above any other traffic on our Token Ring.
- Adding packet priority within the Token Ring device driver. CTMSP uses a packet priority above both ARP and IP packets.
- Splitting out the function that computes the Token Ring header. This allows for precomputing the header once for the life of the connection.
- Adding code to the split point of ARP and IP packets in order to split out the CTMSP packets and correctly handle them.

We implemented these changes in the prototype system for data transport using CTMSP on a Token Ring network.

---

[4]As long as the destination was on the same physical Token Ring.

[5]If we did not do this then we would have the additional problem of creating a router that could keep up with the data rates that we were using. This is possible but has not been implemented.

4. Adapters and CTMS

Current adapters load the CPU too heavily. There are several sources of this load. If the adapter is capable of DMA and the DMA is done into system memory, this DMA can interfere with the CPU's access to system memory. A second source of loading comes from most adapters' use of interrupts. Once the CPU is interrupted, the amount of delay between the start and end of servicing the interrupt is completely dependent on the implementation of the interrupt handler. If this implementation includes long sections of protected code, the problem becomes more acute.

Critical code segments cause another problem with implementation modifications. Given that we want to have interaction of device drivers at interrupt handler execution time, we must protect the critical sections at the highest possible level. By introducing another source of interrupts, the interactions in the UNIX kernel can be nearly impossible to track.

A shortcoming of current Token Ring hardware is its inability to give back an interrupt when a Ring Purge is detected on the network. This leads to the sole source of dropped packets for which no correction can be made. To be able to correct for this, the Token Ring adapter would have to be put in a mode to read all Medium Access Control (MAC) frames that are seen on the ring[6]. This adds yet another loading to the system's ability to respond to interrupts. From observation, the amount of MAC frame traffic on the Token Ring we use is between 0.2% and 1.0%. The MAC frame packets are on the order of 20 bytes of data. Given a 4Mbit Token Ring, there would be between 50 and 250 interrupts to handle MAC frames per second. This additional interrupt and software decoding of packet headers would add an unacceptable amount of overhead to detect the small number of Ring Purges that occur on the Token Ring. Other than this single source of dropped packets, we found that the Token Ring was completely reliable for sending packets between two machines on the same ring.

On IBM RT/PC machines, we can make a change to reduce the loading on the CPU. That is to modify the Token Ring device driver so that it does not use system memory for the fixed DMA buffers. This change is specific to the architecture of the IBM RT/PC. This architecture has two separate buses that transfer address and data information within the machine. The first is between the CPU and the main system memory. The second, normally called the Input/Output (IO) Channel Bus, interconnects all of the attached adapters. The arbiter for accesses between these two bus structures is the Input/Output Channel Controller (IOCC). An adapter is available that is solely memory, called IO Channel Memory. DMA between another adapter and IO Channel Memory can occur on the IO Channel Bus and not cause interference with accesses by the CPU to main system memory. We implemented this change to the Token Ring device driver to use IO Channel Memory in order to reduce the loading on the CPU in the prototype system for data transport using CTMS.

If adapters were designed and manufactured to do data transfers using DMA or directly transfer data between two devices, the load on CPU could be reduced further.

5. Measurement Tools

Once we built the prototype system, we needed tools to measure both the packet activity on the network, as well as several points with the kernel. Commercial products exist that measure the load on a Token Ring and capture packets for later examination quite well. After examining several, we used IBM's Trace and Analysis Program (TAP). This tool allowed for the recording and time stamping of all packets seen on the network, including all MAC frames. The tool also recorded the first Token Ring adapter's buffer of actual packet data (up to 96 bytes) as well as the Token Ring's Access Control byte, Frame

[6]This completely ignores the fact that to get a Token Ring adapter that would pass up the MAC frames requires proprietary software in the ROMs on the adapter.

Control byte and total length. However, there are limitations of the tool's ability to record all packets[7].

Using the TAP tool, we were able to detect when packets were out of order and lost. In the first case, out of order packets were a direct result of the Token Ring device driver implementation. Once the critical sections of code were more carefully protected, the problem of out of order packets completely disappeared. Thus, we found that we were working with a network that would transmit packets in order and do so reliably with only one exception: Ring Purge on the Token Ring.

Ring Purges occur on the network primarily due to new stations inserting into the network or old stations reinserting into the network. If a packet is being transmitted at the time of insertion, it is possible that the packet will be lost. Ring Purge is transmitted by the Token Ring's Active Monitor after such an event has occurred[8].

If the adapter is programmed to interrupt the transmitter when a Ring Purge is seen on the ring, then the transmitter can attempt to correct for a possible lost packet by retransmitting the last packet that is still in the fixed DMA buffer. The receiver, in this case, might need to ignore a duplicate packet if the transmitter incorrectly retransmits a packet. Unfortunately, the adapter does not interrupt the processor with the information that a Ring Purge has occurred. As was mentioned earlier, the only way to detect this occurrence would be to set the adapter to receive all MAC frames and pass them on to the interrupt handler. But, the software on the adapter does not allow for passing MAC frames to the host processor. Even if the software did allow for this, the overhead in handling interrupts and parsing MAC frames to detect a Ring Purge would be unacceptable. We decided to allow for the loss of a single packet and to measure the frequency of this occurrence. Measurement revealed that Ring Purges are normally a result of a station insertion, and occur on the order of 20 times a day. We decided that we could safely ignore this level of lost packets by adding code to recover.

Even though the TAP tool was very useful for the macro scale measurements of the Token Ring, it was not sufficient to measure the actual device driver. We were interested in gathering events and time stamping them with 100 microsecond accuracy. We needed to be able to correlate the time stamps of several other points within the device driver, on possibly more than one machine simultaneously, to determine the correct transmission and reception of packets. These types of micro scale measurements were needed to find the worst case delay of packets as well as the mean and standard deviation of inter-packet departure and arrival times. The available commercial products fell far short of being able to do all of this. The most critical part missing was a time stamping facility with the accuracy required.

5.1. Source of CTMS Data

At this point, we need a short discussion of the source of data for the following measurements. What we required was a stable source of interrupts. We used IBM's Voice Communications Adapter (VCA). Briefly, the adapter has a TI32010 DSP, 2k by 16 bit memory, which is byte accessible by the host processor, can be interrupted by the host and can interrupt the host. We created a program to run on the adapter that would interrupt the host every 12 milliseconds. We added several *ioctl* calls to set up the device in this special mode, to request the Token Ring header and keep this header as part of the state of the device, and to request handles to functions needed by the modified Token Ring device driver. We hard coded in the VCA's device driver calls to the Token Ring device driver for calculation of the Token Ring header and for the sending of a packet.

We modified the VCA's interrupt handler to create a CTMSP packet by allocating a linked list of

[7]For specific details, please see the documentation of the product [IBM90].

[8]For specific details on this, please see [IBM89].

mbufs for the packet and then copying the static precomputed Token Ring header, a destination device number, and a packet number into the packet. We then appended the packet with data to create a packet of 2000 bytes in length (including the header information but excluding the Token Ring protocol bytes). We then sent this packet via the modified Token Ring device driver. The result of this modification was to create a CTMSP data transport stream of approximately 150KBytes/sec.

5.2. Tools Used and Built to Measure the Modified System

After discovering the lack of measurement tools that existed to measure device driver performance on multiple machines at the same time, we decided to look into building our own tools. We used several methods to measure both the VCA and the Token Ring device drivers. We will discuss the error introduced by the various methods of measurement within the description of each method. The points of measurement within the total system were as follows:

- The Interrupt Request Line (IRQ) of the VCA adapter
- Entry into the VCA's interrupt handler
- Immediately after the packet is copied into the fixed DMA buffer and immediately before the Token Ring adapter is given the *transmit* command.
- Immediately after the received packet is determined to be a CTMSP packet.

Given that we wanted to measure these points, what we needed was to add extremely small pieces of code in the appropriate places that would cause the events to be time stamped and recorded.

5.2.1. IBM RT/PC Used as Measurement Tool

We made the first attempt at time stamping events by using a pseudo device driver. We modified the Token Ring device driver to call a procedure within this pseudo device driver. A UNIX *open* call to this device set a flag in the Token Ring device driver that enabled these event time stamping procedure calls. We added an *ioctl* call within the pseudo device to read out the recorded data. We could only measure the last three points mentioned above since this was all done in software.

We were able to coordinate the activities of the transmitter, receiver and the TAP tool under a centralized control point. The end result was a set of computers that recorded and analyzed data in real time. If a packet was lost, had an extremely long inter-departure or inter-arrival time, or there was an incorrect ordering of packets on the transmitter and/or receiver, all machines were halted and a snapshot of the data was taken. We then examined the snapshots to decide what error had occurred. Histograms as well as means and standard deviations were computed for the inter-packet departure and arrival times from this data.

The main problem with recording data using this method was the potential for interaction between the data recording mechanism and the rest of the activity of the machines. The error introduced in the time stamps due to this method was a direct result of this interaction. If the time stamping procedure was done with interrupts disabled, then the procedure itself might delay unacceptably another point of measurement. If interrupts were enabled during the procedure, then the time stamp could be significantly in error due to the possibility that another interrupt occurred while executing the recording procedure. In addition, the clock granularity was only 122 microseconds. All in all, this was a poor method of recording data on inter-packet arrival and departure times, but was extremely good at helping to find bugs in the Token Ring device driver, bugs in the system as a whole, as well as debugging our modifications to the system.

5.2.2. Logic Analyzer and Oscilloscope as Measurement Tools

The use of a logic analyzer is the least obtrusive way of measuring the values of interest, since we could program the analyzer to trigger on any memory read or write access or on any edge of any signal. We used this to determine the degree of accuracy of the VCA interrupt source. The result was that the VCA could interrupt the host processor every 12 milliseconds as desired with no detectable variation. We made further measurements using an oscilloscope to look at the second pulse of the Interrupt Request (IRQ) line given that we were triggering on the leading edge of the first pulse. The second pulse varied on the order of 500 nanoseconds from 12 milliseconds. We considered this conclusive proof that the VCA interrupt source was completely solid and that we need not consider that there was any error introduced by the source of interrupts.

The second use of the logic analyzer was to measure the variation between the IRQ pulse and the start of the VCA interrupt handler. Even while loading the Token Ring and the local disk, the largest variation seen was 440 microseconds.

We discarded using the logic analyzer to measure any of the other parameters due to the limitations of the device and our own expertise. Specifically, we needed a complete histogram of all of the intervals described above so that the total shape of the histogram curves could be examined. The logic analyzer was not capable of this functionality.

5.2.3. IBM PC/AT Used as Measurement Tool

Figure 5-1 shows how we delegated the time stamping of events to a pair of external machines. We used an IBM PC/AT machine with a parallel interface board consisting of eight separate 8-bit wide interfaces. We installed a serial/parallel interface board in each machine on which we wanted to time stamp events. Within the Token Ring device driver, we replaced the calls to the pseudo device driver procedure with in-line code to write specific values into the parallel port and toggle the strobe output line. In the case of transmission or reception of packets, we devised the shortest possible test to determine if the packet was an CTMSP packet. If so, the last 7 bits of the packet number were written to the parallel port and then the strobe output line was toggled.

We then collected the data on the PC/AT within an interrupt handler infinite loop. This loop time stamped the data with a 16 bit clock value where the resolution of the clock was two microseconds. We used another timer within the PC/AT to generate a signal with a period of 50 Hz. We then tied this signal to the eighth parallel input port. This guaranteed that the programs that later analyzed the data could correctly detect the roll over of the 16 bit clock time intervals, even if no other datum occurred between two roll over periods of the clock.

Within the interrupt handler infinite loop, we polled the register that held the bits indicating which channels had an interrupt pending. If any bit was set, we read the clock as well as those ports that had interrupt bits set (excluding the eighth port). We then queued the value of the interrupt register, along with 16 bits of clock time and the values of the ports which were read. If no bits were set in the read of the interrupt register and if there was any queued data, the data was sent out a separate parallel port to a second PC/AT machine. This output to the second machine was fully handshaked. We saved the data received by the second PC/AT over this parallel connection onto a local disk of the second machine.

To determine the error introduced by this method of measurement, we measured the worst case time for the execution of the interrupt handler loop. We found this number by pulling out the VCA's Interrupt Request Line and time stamping occurrences of these events. By using a logic analyzer, we determined that the VCA was able to dependably interrupt on 12 millisecond boundaries within negligible error. Therefore, any variation was due to the measurement tool. Upon examining the spread of the histogram curve generated by this test, we found that there was a 120 microsecond spread on both sides of the 12 millisecond mean. We conducted a second test by using the logic analyzer to measure the loop execution

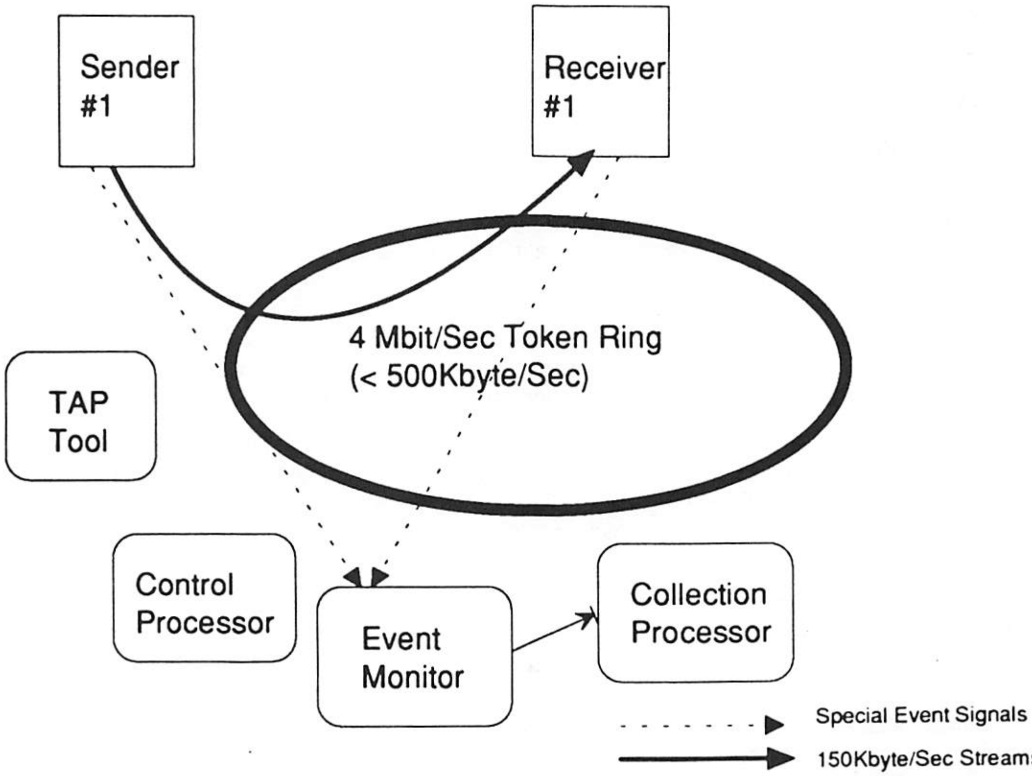


Figure 5-1: Machine Interconnections While Using PC as Measurement Tool

time in the best and worst case conditions. The results were that the interrupt handler loop had a 60 microsecond worst case execution time. Considering both sets of measurements, we concluded that the tool had acceptable error to measure the rest of the system.

5.3. Measurements of the Modified System

Given all of the errors, we wondered at times if we would be able to say anything about what we were doing beyond the obvious; "Sending this amount of data is hard and one must be very careful." We decided that we could.

In all cases, we used a point-to-point static network connection between the transmitter and receiver. Beyond the measurements discussed earlier, the following differences will alter the results:

- Transmitter uses IO Channel Memory vs. System Memory for fixed DMA buffers
- Transmitter copies only header into fixed DMA buffer vs. copying both header and data
- Transmitter copies data from the VCA device buffer to *mbufs* vs. direct copy of data from the VCA device buffer to fixed DMA buffers
- Receiver copies header and data from a fixed DMA buffer into *mbufs* before passing to the VCA device vs. VCA examining the packet while still in a fixed DMA buffer
- Receiver copies data out of *mbufs* into the VCA device buffer vs. no copy of the data (dropping the packet)
- Use of priority within the Token Ring device driver vs. use of same level of priority as all other packets being sent by the local machine
- Use of priority on the Token Ring vs. use of the same level of priority as all other packets on the ring
- Method of measurement: Local (RT/PC), remote (PC/AT), monitoring of network (TAP),

logic analyzer

- Private vs. Public Network
- Level of background load on network
- Transmitter/Receiver in stand alone vs. multiprocessing modes

We selected the following scenarios for the presentation of the data in this paper:

Test Case A) Transmitter uses IO Channel Memory for fixed DMA buffers; transmitter copies both the header and data into fixed DMA buffers; transmitter does not copy data from VCA device into *mbufs*; receiver copies data from fixed DMA buffer into *mbufs*; receiver does not copy data from *mbufs* into the VCA device buffer; priority within the Token Ring device driver of CTMSP packets; Token Ring priority; remote measurement; private network; no loading of network; transmitter and receiver in stand alone mode.

Test Case B) Transmitter uses IO Channel Memory for fixed DMA buffers; full copying of data on Transmitter and Receiver; priority within the Token Ring device driver of CTMSP packets; Token Ring priority; remote measurement; public network; normal loading of network; transmitter and receiver in multiprocessing mode but not heavily loaded.

In both cases, we examined histograms of the following measurements:

1) The inter-occurrence of pulses of the VCA's Interrupt Request Line

2) The inter-occurrence of the entry into the VCA's interrupt handler

3) The inter-occurrence of the point immediately after the packet is copied into the fixed DMA buffer and immediately before the Token Ring adapter is given the *transmit* command

4) The inter-occurrence of the point immediately after the received packet is determined to be a CTMSP packet

5) The differences between like occurrences of (1) and (2)

6) The differences between like occurrences of (2) and (3)

7) The differences between like occurrences of (3) and (4)

Test cases A and B, histograms 1 through 5 as well as test case A, histogram 6 all showed values which could easily be explained given the total system and its interactions. Test case B, histogram 6 is shown in Figure 5-2. This particular histogram is interesting because of the bi-model curve. The explanation for this is simply that there is interaction between the transmission of CTMSP packets and the transmission of other system packets. The other traffic includes AFS *keep alive* packets, ARP traffic and socket *keep alive* packets. The socket packet traffic is an artifact of the test set up. All of the machines in the test are being directed by a central control machine. The communications link between the control machine and each of the other machines in the test is via UNIX sockets. All of this system traffic causes some of the CTMSP packets to be queued rather than sent immediately. The system then plays catch up for tens of CTMSP packets. There are 68% of the data points within 500 microseconds of 2600 microseconds, 15% within 500 microseconds of 9400 microseconds, 16.5% between 2800 and 9300 microseconds with the remaining 2% in the tails of the graph which extend from 100 microseconds to 14000 microseconds. The 2600 microsecond mean of the first peak in the histogram gives the mean latency of sending a 2000 byte packet. The transfer rate of copying data from the system memory where the *mbufs* are located to the IO Channel Memory, where the fixed DMA buffers are located, is on the order of 1 microsecond per byte. This leads to 2000 microseconds of latency specifically attributable to copying the packet. The additional 600 microseconds can be attributed to the execution of the code between the two points of measurement.

Test case A, histogram 7 is shown in Figure 5-3. This graph shows that the minimum latency of a 2000 byte packet is 10740 microseconds. Specifically, 98% of the data points fall within 160

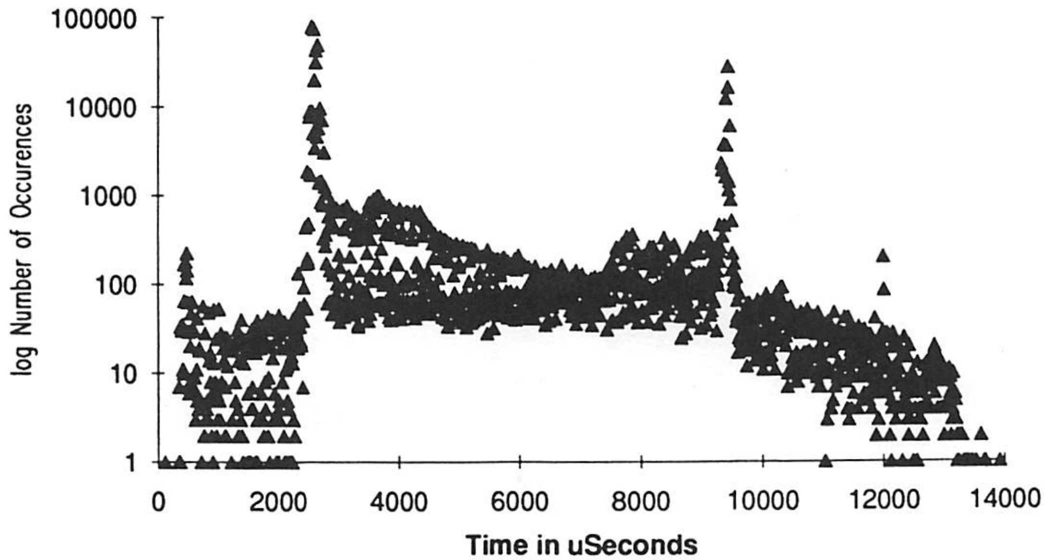


Figure 5-2: VCA Interrupt Handler Entered to Just Prior to Transmission

microseconds of the 10894 microsecond mean with the remaining 2% spread to the right of the mean extending to 14600 microseconds. The latency seen can be attributed to DMA'ing the data on both ends, transmission time across the Token Ring, delay of the receiver's interrupt handler being entered and execution of the receiver's interrupt handler code. The spread of the curve can be attributed to other interrupt sources and the execution of protected code segments throughout the kernel. Additionally, there is delay if a packet is on the Token Ring at the time of transmission. The only type of packet other than the CTMSP packets in this test case is the normal MAC frame traffic, which uses 0.2% of the network in this completely unloaded test case.

Test case B, histogram 7 is shown in Figure 5-4. This graph shows that the minimum latency of a 2000 byte packet is 10750 microseconds. Specifically, 76% of the data points fall within 160 microseconds of the 10900 microsecond peak, 21.5% fall within the range from 11060 to 15000 microseconds, 2.49% fall within the range from 15000 to 40050 microseconds, with the remaining two points falling in the range from 120 to 130 milliseconds. These last two exceptional points are not in the histogram as shown. Examining the normal traffic on the network shows that there are three different sizes of packets. The first size consists of MAC frame packets of approximately 20 bytes in total length. The second size consist of ARP packets and those packets which comprise the *keep alive* packets for AFS. Each of this second class of packets are approximately 60 to 300 bytes in total length. Lastly, there are the file transfer packets sent while a compile is done or during UNIX kernel copying activity. These packets are 1522 bytes in total length. The traffic associated with these three sets of packets can justify some of the spread of the curve as well as some of the secondary peaks. Most of the remaining points can be attributed to the interaction of the reception and transmission of packets other than the CTMSP packets. The two exceptional points in the range of 120 to 130 milliseconds can not be completely accounted for. We can only speculate that a large number of circumstances occurred at the same instance. The majority of this time can be attributed to both a soft error on the Token Ring and the Token Ring timing out and resetting

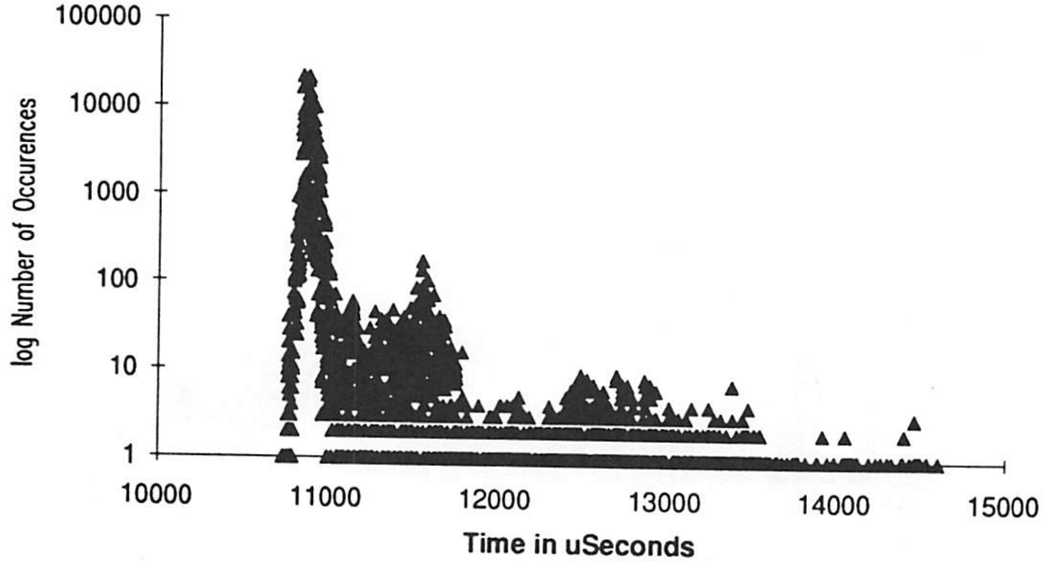


Figure 5-3: Transmitter to Receiver Times, Test Case A

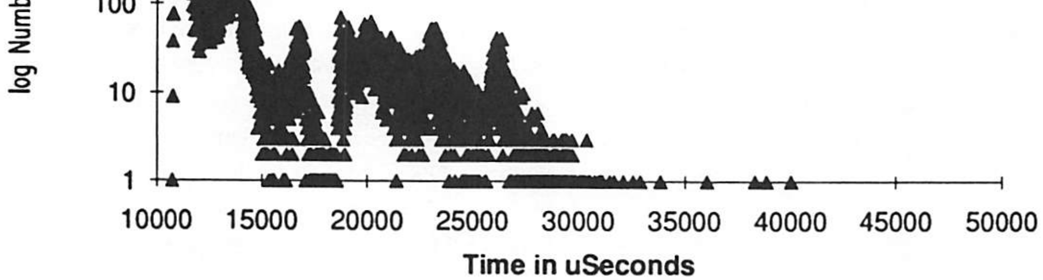


Figure 5-4: Transmitter to Receiver Times, Test Case B

of the network. Unfortunately, this can only account for 10 milliseconds. If we speculate that a ring insertion would cause multiple Ring Purges, then we could completely account for this activity. Experimentally, we have seen on the order of 10 Ring Purges back to back. We conclude that this is precisely what is occurring. We then measured the number of insertions seen in one day. The number was under 20, approximately one an hour. The test data shown for Test Case B was run for 117 minutes. We conclude that the two exceptional data points are two insertions into the Token Ring.

6. Conclusions

In order to transport 150KBytes/sec of CTMS data, two modifications are necessary. The UNIX model of device to device data transfers must be changed to eliminate a minimum of two data copies. This can be done by transferring the data directly between two devices rather than indirectly via a user process. Secondly, a new network protocol must be used. It should be noted that the intent of this work was not to define the architecture of this new protocol but rather to build a prototype system that could be measured to help with the later definition of the protocol. Finally, a third modification is useful. This third modification is the use of IO Channel Memory for the fixed DMA buffers.

As for conclusions on the measurement of the prototype system, the worst case times between transmission and reception of a single packet is 40 milliseconds. There are two exceptional data points within the 120 to 130 millisecond range. Both of these points are explained by the Token Ring timing out and resetting itself during a ring insertion or reinsertion by a station. Even with these exceptional data points, the buffer space needed for 150KBytes/sec CTMSP data transfer is under 25KBytes. This amount of buffer space is well within a reasonable range to support the functionality of data transport of Continuous Time Media Systems.

References

Comer88. D. E. Comer, Internetworking with TCP/IP: Principles, Protocols, and Architecture, Prentice Hall, Englewood Cliffs, NJ, 1988.

IBM89. IBM Token Ring Network Architecture Reference, SC30-3374-02, third edition 1989.

IBM90. IBM Token Ring Network 16/4 Trace and Performance Program User's Guide, 93X5688, first edition June 1990.

Leffer89. Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, John S. Quarterman, The Design and Implementation of the 4.3BSD UNIX Opertating System, Addison-Wesley Publishing Company, 1989

Morris86. James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, F. Donelson Smith, Andrew: A Distributed Personal Computing Environment, Communications of the ACM, Volume 29, Number 3, March 1986.

Stevens90. W. Richard Stevens, UNIX Network Programming, Prentice Hall, 1990.

Tevanian87A. Avadis Tevanian, Jr., Richard F. Rashid, Michael W. Young, David B. Golub, Mary R. Thompson, William Bolosky, Richard Sanzi, A Unix Interface for Shared Memory and Memory Mapped Files Under Mach, Carnegie Mellon University, Department of Computer Science, July 1987.

Trevanian87B. Avadis Tevanian, Jr., Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach, Carnegie Mellon University, Department of Computer Science, CMU-CS-88-106, December 1987.

TI88. First-Generation TMS320 User's Guide, Technical Report, Texas Instrutments.

Biographies

Michael Pasieka received his BS degree in Computer Science and MS degree in Electrical Engineering and Computer Science from MIT in June, 1984. He has worked for Texas Instruments, Amdahl, Symbolics and is currently working for the Information Technology Center at Carnegie Mellon University. He has specialized in inter-computer communications, working on such projects as serial links, input/output testing for Amdahl class machines, and currently transport of data at a sustained high rate in support of distributed multimedia.

Paul G. Crumley has worked in the field of computing systems for about 15 years. Though he has an Electrical Engineering degree from Carnegie Mellon University (BS EE 1983) he can appreciate a well designed and implemented piece of software. Paul has been a member of the Information Technology Center at CMU for seven years where he currently leads the Continuous Time Media System group. Paul is interested in architectures for parallel processing of data, the implementation of distributed systems and photography.

Ann Marks received her BS in Electrical Engineering in May, 1976, Masters of Engineering (Electrical) in May, 1977, and Ph. D. Electrical Engineering/Computer Science in August, 1980, from Cornell University. She is a member of IEEE and ACM and has worked at the Information Technology Center at Carnegie Mellon University since July 1987. Her current research interests are the transport of continuous time media and server design for continuous time media. Prior work includes document interchange using ODA for the Expres Project. She is a co-author of a book on document interchange entitled *Mulit-media Document Translation ODA and the Expres Project* (Springer-Verlag, 1991).

Ann Infortuna received a BS degree in Computer Engineering from Lehigh University, in 1984, and an MS degree in Electrical Engineering from the University of Rochester in 1988. She previously worked in system design and development of new products for the Xerox Corp., and is currently working for the Information Technology Center at Carnegie Mellon University where she is invovled with the design of an execution environment for distributed multimedia.

Multimedia/Realtime Extensions for the Mach Operating System

Jun Nakajima, Masatomo Yazaki, Hitoshi Matsumoto

Human Interface Laboratory
Fujitsu Laboratories, LTD.

{nakajima, yazaki, matumoto}@flab.fujitsu.co.jp

Abstract

We have extended the Mach operating system to support multimedia applications. Our extensions for Mach provide the system designers and the developers with a reliable and flexible multimedia processing environment. This paper focuses on realtime issues while supporting multimedia applications, asynchronous event notification which follows the POSIX P1003.4 proposal, preemptive deadline scheduling, and device drivers which run in user mode. Preemptive deadline scheduling preempts asynchronous event notification for continuous media that requires each event handler to execute as soon as possible and schedules other event handlers to meet their time constraints as well. The system designers must develop device drivers for a wide variety of multimedia devices. New devices will appear as multimedia technology progresses. User-mode device drivers will help develop device drivers efficiently and provide a flexible environment. The rationale, design, implementation and performance are presented.

1 Introduction

The motivation for the development of this multimedia/realtime extensions has arisen from involvement with an attempt to port where we moved some multimedia applications — a hypermedia authoring system and an interactive music system [1] — from MS-DOS to UNIX/Mach[2].

A serious problem encountered during the porting attempt was due to the implementation of multimedia handlers having realtime constraints. For example, the interactive music system requires the MIDI(musical instrument digital interface) handler to output MIDI commands at specific intervals; if the handler outputs delayed commands for a note or skips one, the music will stumble. The user-mode handler computes the commands dynamically. Hence we need a periodic event notification facility to implement such an application. As described in an IEEE proposal [3], there are deficiencies in the traditional UNIX signal mechanism. We should use asynchronous event notification for this purpose. The priority scheduling under the proposal, however, lacks predictability. It forces system designers to map a set of specified realtime constraints into a priority order in such a manner that all event handlers will meet their deadlines [4].

Another problem with the porting attempt was to implement and integrate modern device drivers for multimedia. The devices need interfaces that synchronize with each other, and that have realtime constraints. In addition, the system designers must develop device

drivers for a wide variety of multimedia devices. New devices will appear as multimedia technology progresses. A user-mode device driver is implemented by using asynchronous event notification generated by a multimedia device, and is scheduled by preemptive deadline scheduling. User-mode device drivers will help designers develop device drivers efficiently and provide them with a flexible environment.

The Unix Server [5] shows that it is both possible and practical to implement UNIX as an application program. This fact partly motivated us to implement the user-mode device drivers. Compared with the traditional implementation, user-mode device drivers have the following advantages:

- schedulability – in a traditional implementation of UNIX, the current interrupt priority level determines whether an interrupt is blocked. Hence, it is impossible to schedule each device drivers according to the scheduling policies.
- extensibility – system designers can develop and install new device drivers without rebuilding the kernel.
- flexibility – to improve performance or to simplify application program code, a multimedia device drivers can be replaced for a specific purpose.

We chose to use the Mach operating system for our implementation because:

- The Mach thread model and portable implementation made the implementation easier than it would have been under a traditional UNIX.
- Multimedia application may require a multithreaded implementation model[6].

The Mach 2.5 scheduler, however, only has round-robin and fixed priority policies managing 32 levels of thread priorities[7]. To support threads that have realtime constraints, we need other scheduling policies that distinguish between realtime and non-realtime threads[8]. *Preemptive deadline scheduling* preempts asynchronous event notification for continuous media that requires each event handler to execute as soon as possible and schedules other event handlers to meet their time constraints as well.

This paper describes our implementation of multimedia/realtime extensions for the Mach operating system. Section 2 summarizes POSIX asynchronous event notification and defines an interrupt handler in our extensions. Section 3 discusses the realtime constraints in multimedia applications. Section 4 describes the extension kernel and scheduling policies for multimedia processing. Section 5 describes user-mode device drivers. Section 6 presents performance measurements from several experiments, and section 7 represents an overview of related works. Finally, section 8 presents our conclusions.

2 Asynchronous Event Notification

Multimedia applications are implemented using integrated and coordinated processing by the multimedia handlers. System designers and multimedia developers assume that a multimedia application will require the operating system to support event notification (interrupts which are generated by the multimedia devices). In addition, they assume that processes will require the event handlers to perform asynchronous operations in parallel and in a realtime manner, if necessary. A traditional UNIX provides the signal facilities for event notification, but they are not suitable for such multimedia processing: the signal mechanism lacks deterministic execution of signal handlers, the set of user definable signals is limited, the latency of signal delivery is unpredictable, and signals can be lost[3]. Because

of the high portability of Mach, we should improve the portability of device drivers that are believed to be very machine-dependent.

POSIX asynchronous notification is a determinism enhancement facility replacing the signal mechanism to make asynchronous notifications to an application be queued without impacting compatibility with the existing signals interface. Events represent occurrences that are generated as the result of some activity in the system. We added interrupts from multimedia devices to the set of events addressed in [3].

The interrupt handler for a multimedia device is a thread that executes an *event notification function*. An event notification function is a function that is defined in an application program and is executed upon asynchronous event notification. A thread that executes an event notification function is called a "handler". Unlike ordinary event notification functions, an interrupt handler needs access to the hardware device. Section 5 discusses these requirements.

3 Realtime Threads

The following subsections present the decisions we faced when designing realtime threads to meet realtime constraints in multimedia and user interfaces to defining an event and an event notification function.

3.1 Realtime Constraints in Multimedia Applications

Multimedia applications may require event handlers to meet realtime constraints because:

- Some media, such as sound and music, are continuous.
- Some media need to synchronize accurately with a hardware device whose status is unknown except for the time elapsed.

We divided multimedia devices into two types, *deadline-driven* and *event-driven*, according to their realtime constraints. A *deadline-driven device* requires both its operations and the arrival of data to meet a deadline. If the operations to the device, including the transference of data, are completed before a deadline, the media never deteriorates. A typical deadline-driven device is a graphic display. A *event-driven device* requires its operations to execute immediately after the occurrence of the event. The device may have a deadline as well. The media may deteriorate as the response time increases. A typical event-driven device is a MIDI device. Any multimedia hardware device whose status is unknown, except for the time elapsed since it started, is considered as an event-driven device. Figure 1 shows examples of deadline- and event-driven devices.

To support multimedia applications which use these types of devices in parallel, a proper scheduling policy is to preempt handlers for the event-driven device handlers and schedule the deadline-driven device handlers to meet their deadlines.

3.2 Creating and Terminating Realtime Threads

An event notification function is implemented as a *realtime thread* in our implementation. To meet the constraints discussed above, the realtime threads are divided into two types: *standard* and *pressing*. A *standard realtime thread* only requires that it should meet the deadline; a deadline-driven device driver is implemented by a standard realtime thread. A *pressing realtime thread* requires that it should be executed promptly and meet the deadline as well; a event-driven device driver is implemented by a pressing realtime thread.

Device	*Media*	*Device type*
graphic display	animated graphics	deadline-driven
video recorder	video	event-driven
CD player	music	event-driven
FM sound source chip	voice, music	event-driven
MIDI	music	event-driven
PCM sound source chip	voice	deadline-driven

Figure 1: Multimedia device types

Realtime threads are also defined as the periodic or one-shot according to their functions. This is set in a structure specifying realtime constraints of the event notification function.(See Section 3.4)

3.2.1 Creating Realtime Threads

A realtime thread, within a task, is created by the *cthread_fork*() function [9] as an ordinary thread which is managed by the original Mach kernel. A call to *evtenter*() places the thread on the *event waiting queue*, changing the state in the thread structure to "realtime". Then the realtime thread is suspended, awaiting the occurrence of the events that are of concern to it. A thread in the "realtime" state is managed by the extension kernel, which is discussed in Section 4. The realtime thread is resumed by occurrence of the events that are of concern to it, and it runs in user mode and returns to the kernel by calling *evtreturn*(). The function operates like the *sigreturn*() system call in the implementation of signals. If the thread is defined as periodic, the *evtreturn*() function replaces the thread on the event waiting queue. Otherwise it terminates by the *exit*() system call.

3.2.2 Terminating Realtime Threads

A realtime thread terminates like an ordinary thread; it terminates either voluntarily through an *exit* system call or involuntarily as the result of the task termination. Within the kernel, a realtime thread terminates by calling the *exit*() routine like ordinary threads. To halt a realtime thread, extra operations are needed in addition to those for ordinary threads. The steps involved in terminating a realtime thread are as follows:

1. Cancel any pending events that are of concern to it.
2. Neutralize the device table if the thread is an interrupt handler. This is because the thread may have started a device and been waiting.
3. Reset the thread state to the "ordinary".

The extensions provide realtime threads with the *evtrenouce*() function to avoid their monopolization of the processor; a call to *evtrenouce*() resets the "realtime" flag in the state of the thread structure, and the thread is then managed by the original Mach kernel until it calls *evtreturn*().

3.3 Memory Management

Because page fault handling causes unpredictable delays, specific regions of the address space need to stay resident for a realtime thread. RT-Mach [8] provides *vm_wire*() for this purpose; the primitive can "pin-down" a specific region of its parent task's virtual address space. The current version of our extensions provide the same primitive.

3.4 Realtime Thread and Event Definition

Each realtime thread Th_i is characterized by the following parameters in the kernel:

- start time S_i
- deadline D_i by which Th_i must be completed
- estimated worst case execution time C_i
- thread-type flag F_i to indicate whether Th_i is standard or pressing
- weight W_i which determines the relative importance of Th_i among all the threads.

This subsection describes the POSIX event definition and our extensions that are provided for application programs. The event definition *event structure* or **struct event** is given in Figure 2. The *evt_handler* element is a pointer to an *event notification function.* The

void	(**evt_handler*)();
void	**evt_value*;
evt_class_t	*evt_class*;
evtset_t	*evt_classmask*;
evtopt_t	**evt_option*;

Figure 2: Event definition structure

evt_value element is an application- dependent value to be passed to the application at time of event notification. The *evt_class* specifies the event class for the event. The *evt_classmask* specifies the set of event classes blocked if and when the event notification function executes [3].

The *evt_option* specifies the realtime constraints for the event notification function. This is an extension. The definition is the *event option definition structure* or **struct evtopt** is given in Figure 3.

timespec_t	*evt_dtime*;
timespec_t	*evt_etime*;
int	*evt_weight*;
evt_type_t	*evt_type*;

Figure 3: Event option definition structure

The *evt_dtime* specifies the time interval between an occurrence of the event and the deadline. The *evt_etime* is an estimate of the worst case execution time. The flag *evt_type* specifies the event type and is described in Figure 4.

C_i, F_i and W_i are given by this structure. D_i is calculated by adding *evt_dtime* to the time that the event occurs.

3.5 Implementation of mkevent()

The POSIX function *mkevent*(), that creates and initializes an event definition, can be implemented as in Figure 5.

event type	*description*
ET_PPER	pressing and periodic
ET_PONE	pressing and one-shot
ET_SPER	standard and periodic
ET_SONE	standard and one-shot

Figure 4: Event types

```
#include        <cthreads.h>
#include        <sys/events.h>
#include        <sys/timers.h>

typedef struct event    event_t;

event_t *mkevent(func, evtvalue, evtclass, evtclassmask, option)
        void    (*func)();
        void    *evtvalue;
        evt_class_t evtclass;
        evtset_t    evtclassmask;
        evtopt_t    *option;
{
        event_t *eventp;
        void    async_handler();

        eventp = (event_t *)malloc(sizeof(event_t));
        if (eventp == (event_t *)NULL)
                return NULL;
        eventp->evt_handler = func;
        eventp->evt_value = evtvalue;
        eventp->evt_class = evtclass;
        eventp->evt_classmask = evtclassmask;
        eventp->evt_option = option;
        cthread_detach(cthread_fork(async_handler, eventp));

        return eventp;
}

void async_handler(eventp)
        event_t *eventp;
{
        evtenter(eventp, (timespec_t *)NULL);
}
```

Figure 5: Implementation of *mkevent*()

4 Multimedia/Realtime extension Kernel

The multimedia/realtime extended Mach kernel consists of the original Mach kernel and the *multimedia/realtime extension kernel.* The multimedia/realtime kernel provides:

- a subset of POSIX asynchronous event notification facilities,
- asynchronous input and output, and
- preemptive deadline scheduling which manages the notification functions.

The multimedia/realtime extension kernel manages a *realtime thread queue*, which contains all the runnable realtime threads, except the currently running thread, sorted by deadlines D_i on the queue.

An event is generated by either a user task or a hardware device by calling the *evtraise*() or *evtgraise*() function, respectively. The definition of the *evtraise*() function follows the POSIX proposal. It generates an application-defined event for the calling task. The kernel function *evtgraise*() generates an event for all the realtime threads that unblock the event in the *event waiting queue.* The awakened realtime threads are placed on the *realtime thread queue* in a manner that all the threads are sorted by their D_i. Preemptive deadline scheduling rearranges the threads on the queue, if necessary.

When a new thread is needed for the execution, the kernel takes the following steps:

1. Saves the current context with a call to *save_context*().
2. If the currently running thread is a *realtime thread,* places it on the realtime thread queue. Rearranges the threads on the queue by preemptive deadline scheduling.
3. If the *realtime thread queue* is empty, the original Mach kernel chooses a new thread.
4. If the queue is not empty, removes the first thread from the queue.
5. Sets the new thread running with a call to *load_context*().

A pressing realtime thread can be considered as a thread whose deadline is imminent. In a single processor operating system kernel, dynamic realtime scheduling, such as deadline scheduling, provides high processor utilization and versatility [10, 11]. It is, however, difficult for a user task to find an appropriate deadline so that the thread is scheduled as soon as possible and meets the deadline.

Preemptive deadline scheduling preempts a pressing realtime thread, while scheduling the other standard realtime threads to meet their realtime constraints. Such a preemption may lower the capacity of deadline scheduling, but the effect will not be serious under the assumption that ordinary event-driven devices, such as MIDI device, have no buffers, or very small buffers, and the handlers can be completed within a relatively short period.

Preemptive deadline scheduling policies have the following characteristics:

- Qualitative control – preemptive deadline scheduling preempts the pressing realtime thread with highest weight on the queue. It provides system designers with qualitative control of media in multimedia applications.
- Guaranteed timing – standard realtime threads are scheduled to meet their deadlines.

Preemption of a pressing realtime thread is not always allowed, because it may cause a timing fault for standard real threads in the queue. Fortunately, time faults for continuous media is often imperceptible, depending on the media and the application. Preemptive deadline scheduling allows system designers to determine which thread, to delay or leave unfinished. W_i is used for this purpose.(See Figure 6)

The pseudo-code for the basic preemptive deadline scheduling algorithm is given in Figure 6. Note that all the threads have been sorted by their D_i on the realtime thread queue and that D_0 is the head.(See Section 4)

5 User-Mode Device Driver

A user-mode device driver, which we can call a multimedia device server, is implemented by using asynchronous event notification by a multimedia device, and scheduled by preemptive deadline scheduling.

In designing these features, it was challenging to define and integrate the new interface between the drivers and the kernel without impacting compatibility with the traditional interface. A traditional UNIX supports two standard interfaces to I/O devices through block and character special device files. The extension kernel provides two internal interfaces that may support a multimedia device driver. These interfaces permit a device to be used in a two ways: as a *deadline-driven device* suitable for completing an operation before the deadline, or as an *event-driven device* suitable for executing an operation immediately after the occurrence of the event.

The following subsections present an interface which supports user-mode device drivers. Figure 7 shows the general architecture of user-mode device drivers.

5.1 Access to User Address Space and Devices

Mach supports *vm_read*() and *vm_write*(), which allows one task to read another's memory and allows task's memory to be written by another, respectively.

Because traditional tasks cannot directly access most I/O devices, it is necessary to provide user-mode device drivers with reliable and efficient access to the devices and buffers (physical memory). The current version of the extension kernel provides user-mode device drivers with system calls for these purposes:

- *dev_mmap*() – establishes a mapping between the task's address space(device data structure) with the device represented by the device number, and
- *dev_mmove*() – transfers the contents of the task's memory to the buffer of the device represented by the device number.

A device data structure is used to refer to the device registers and their fields. (See [12]) The *dev_mmove*() function provides user-mode device drivers with the DMA capability.

5.2 Critical Section and Interrupt Blocking

Some operations on hardware devices may not work unless they are performed continuously. If the processor enters a *critical section*, the thread is cannot be preempted by other threads until the processor exits the section. The *dev_begin_critical*() and *dev_end_critical*() primitives to define a critical section.

An event notification function can set or get currently blocked events using the POSIX *evtprocmask*() function and, hence, it can block all interrupts generated by the traditional UNIX hardware devices, except the hardclock interrupt.

Preemptive Deadline Scheduling
begin
  Search the realtime thread queue for pressing realtime threads ;
  if (there are no pressing realtime threads) **then**
    return ;
  Let P be the set of indexes of the pressing realtime threads on the queue ;
  Find Th_α that has $W_\alpha = \max_{i \in P} W_i$;
  if (Th_α is the head on the queue) **then**
    return ;
  else
  begin
    Let Q be the set of the indexes of the standard threads $Th_k (0 \leq k < \alpha)$ such that
    $C_\alpha > D_k - \sum_{i=0}^{k} C_i - current\_time$;
    if $W_\alpha > \max_{i \in Q} W_i$ **then**
      Move Th_α to the head on the queue ;
    else
      return ;
  end ;
end ;

Figure 6: Preemptive deadline scheduling algorithm

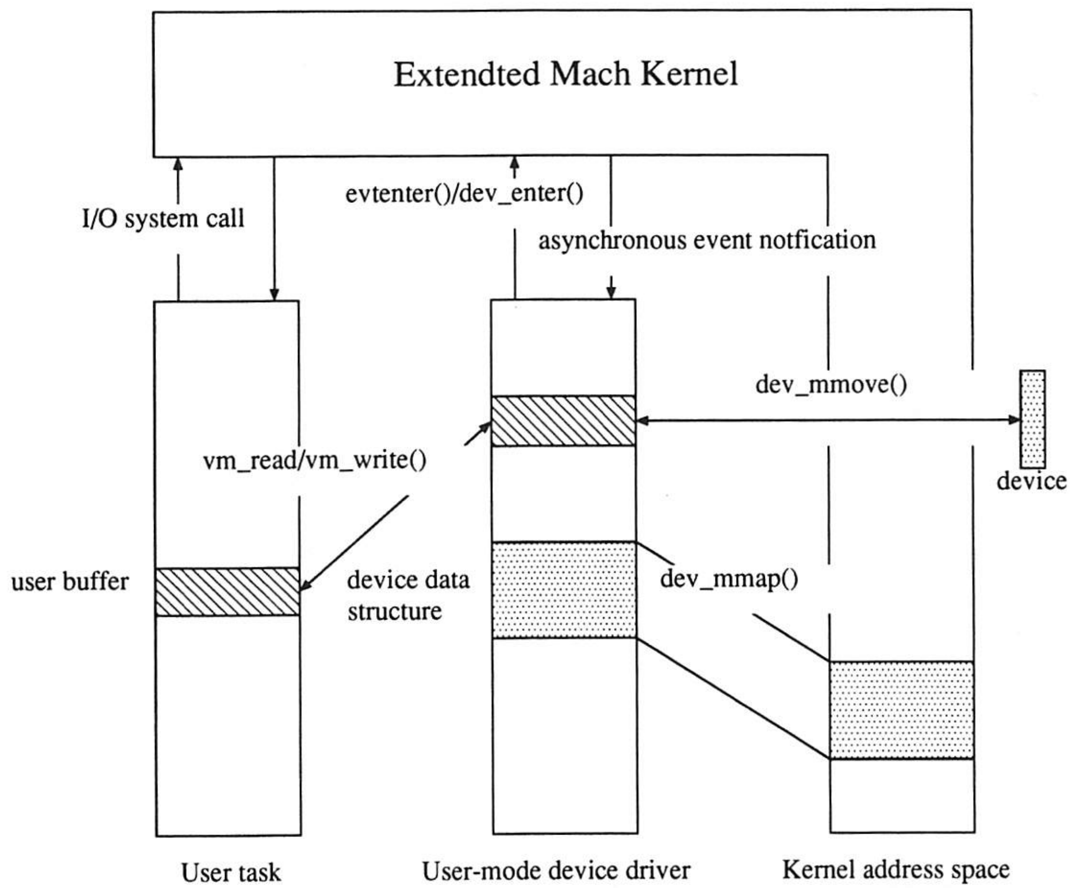


Figure 7: The architecture of user-mode device drivers

5.3 Interrupt and I/O Request Service

Routines in a device driver are divided into two main sections: interrupt service and I/O request service. An interrupt handler for a multimedia device is an event notification handler; it is implemented using asynchronous event notification facilities and critical sections. A user-mode device driver includes realtime threads, or *service threads* corresponding to the service routine for I/O request in the traditional UNIX device drivers, such as *open*(). To allow running as an application program, each multimedia device driver is described by pointers to threads in the *deadline-* and *event-device tables.* Each device driver is described by entry points in the block- and character-device tables in a traditional UNIX.

5.4 Service Thread

A service thread is a realtime thread created by the *cthread_fork*() function. A call to *dev_enter*() places the thread on each entry table(deadline- or event-device table), changing the state in the thread structure to "realtime". Each service thread waits for a request from the kernel. To support the realtime constraints in application programs, the realtime constraints in the event option definition structure are inherited by awakened service threads.

A system call, associated with a multimedia device, wakes up the service thread, then

the thread starts its operation in user mode leaving a *dev_message* value in the entry table. The *dev_message* value is given by *dev_enter*() and corresponds to error codes returned by system calls, such as EBUSY(mount device is busy). Each service is implemented by a *dev_enter*() loop.

The kernel passes the service thread information about:

- the identifier of the task that performed the I/O request, and
- the description about I/O data in the task.

6 Performance

We experimented with asynchronous event notification and user-mode device drivers by modifying the Mach 2.5 kernel to support preemptive deadline scheduling.

6.1 Event Dispatch Latency

Measurements of event dispatch latencies were taken for a MIDI handler in typical UNIX/Mach environments — multiuser operation with several daemons running. An event dispatch latency is the time interval between the occurrence of an event and the execution of the first instruction of the event notification function in response to the event. (See [3]) The MIDI handler is a pressing realtime thread which runs in user mode , and performs a piece of music; *evt_dtime* is 120 ms, *evt_etime* is 5 ms, and *evt_type* is `ET_PPER`.

Each measurement was taken more than 10,000 times on an FM TOWNS with an i386 processor (about 2 MIPS) and 8 megabytes of memory. The purpose of this experiment was to investigate event dispatch latencies under particular conditions. Each of the following six tasks ran in parallel with the MIDI handler

- task #1(spin loop) – continuously spins in a loop.
- task #2(low PCM) – outputs PCM sound. Its weight is smaller than that of the MIDI handler.
- task #3(high PCM) – outputs PCM sound. Its weight is larger than that of the MIDI handler.
- task #4(thread switching) – a multithreaded task which includes ten threads. The threads run in parallel, switching each other by system calls. This task models a task which performs light system calls very frequently.
- task #5(file copy) – copies files by `cp`.
- task #6(kernel build) – builds a kernel object, executing `make`, `sh`, `cc` and etc.

The two PCM tasks are identical except in weights, which determine their relative importance; *evt_dtime* is 150 ms, *evt_etime* is 15 ms, and *evt_type* is `ET_SPER`. The results (in milliseconds) are shown in Table 1. The *Latecomers* are the number of the event dispatch latencies that are longer than 5 ms per 1000 asynchronous event notifications. The tolerable latency depends on the media and applications. An application like Keynote[13] needs a clock accurate to 5 ms or less.

The results of these experiments show:

- short and practical event dispatch latencies can be achieved, except the last two cases, and

- the latencies can be controlled by preemptive deadline scheduling.

Table 1: Event dispatch latencies in a MIDI handler (in milliseconds)

Combination(MIDI +)	*Min*	*Max*	*Average*	*Latecomers*
NULL	0.73	1.71	0.75	0/1000
#1(spin loop)	1.00	3.04	1.20	0/1000
#2(low PCM)	1.00	2.40	1.09	0/1000
#3(high PCM)	1.01	8.49	1.14	8/1000
#4(thread switching)	0.96	3.21	1.44	0/1000
#5(file copy)	1.01	7.98	1.09	7/1000
#6(kernel build)	0.73(0.74)	18.47(7.83)	1.35(1.27)	15(6)/1000

Because the Mach 2.5 kernel is not preempted to run another thread while executing the code to do a system call, asynchronous event notification was sometimes unexpectedly delayed, but it was rarely perceptible to us in these experiments. For example, in the last case, task #6 executes many *fork*() system calls, which take a relatively long time for the kernel to complete. We modified the code of *fork*() so that the kernel can switch to a realtime thread while executing the code. The Table 1 shows the result in the parentheses. However, such modification cannot completely solve the problem; we would need to modify all of the UNIX code.

One of effective solutions to reduce the latencies is to have the kernel switch a realtime thread while executing a code for a system call. Because the Mach kernel supports multiprocessor, a set of system calls can run actually in parallel on a multiprocessor based machine. Thus, the Mach kernel running on a single processor, could be is preempted to run other threads while executing the code to do specific system calls. We thus modified the Mach kernel so that it runs a realtime thread Th while executing the code to do a system call f_K under the condition:

> $f_K \in SP$, the set of system calls which can be executed in parallel, and
> for each f_{Th}, system call executed in Th, $f_{Th} \in SP$.

This modification, however, is very machine-dependent, because we need to implement the AST(asynchronous system trap) in kernel mode. Because of the high portability of Mach, this modification is not desirable. Mach 3.0 or the micro kernel and the Unix Server could provide a more suitable execution environment because we could reduce unexpected delays in the kernel.

6.2 Implementing a Multimedia Application

We attempted to implement a multimedia application, called "Neuro Musician" on our extended Mach. "Neuro Musician" is an interactive musical system; a human performs a piece of music for several measures, then the computer replies with a piece of music based on the input. The music is composed by a neural network which has been taught a certain musical style. The session continues by repeating such musical interactions. The requirements for the implementation are as follows:

- The MIDI handler needs to output MIDI commands(for both notes and timing clock) at specific intervals, and
- The MIDI handler also needs to receive the MIDI commands from the instrument used by the human player while composing the reply or outputting MIDI commands.

- The composition needs to be completed within a specific time to continue the musical interaction between the human and the computer.

Timing clock commands are devoted to synchronization between the musical instruments, and are outputted more frequently (say, at the intervals of 15 ms) than those for notes; six timing clock commands corresponds to the interval of a sixteenth note.

We needed two asynchronous event notifications for input and output MIDI commands. Asynchronous event notification to the MIDI-output handler was implemented by a interval timer. The Mach version of "Neuro Musician" can give more expressive performance (such as graphics which synchronizes with the music) than the MS-DOS version. This is because notification facilities and multithreaded programming are available in the extended Mach environment.

7 Related Work

In this section, we review a few specific systems that have been implemented or proposed. The Unix Server [5] is a practical implementation of UNIX as an application program. Golub et al. have already achieved near parity with Mach 2.5 and can outperform some commercial UNIX implementations. We believe that this suggests that performance of user-mode device drivers is practical.

Real-Time Mach[8] is a realtime version of Mach which can support a predictable and reliable realtime computing environment. Although Real-Time Mach exists, our goals was to provide system designers and the developers with a reliable and flexible multimedia processing environment under ordinary UNIX/Mach environments.

Several algorithms for realtime systems have been proposed or implemented. The priority scheduling which is available under the POSIX proposal, lacks predictability, and the system designers must map a set of specified realtime constraints into a priority order in such a manner that all event handlers will meet their deadlines [4].

Shih et al. [14] considered the problem of scheduling tasks, each of which is logically decomposed into a mandatory subtask and an optional subtask. The mandatory subtask must be executed to completion to produce an acceptable result. The optional subtask begins after the mandatory subtask is completed and refines the result to reduce the error in the result. However, it may be difficult for system designers to decompose multimedia applications into a mandatory subtask and an optional subtask.

The predictive deadline scheduling algorithm [11] extends Liu and Layland's [10] deadline mechanism by requiring system processes to adhere to functional prioritizatoin and maintain dynamic estimates of the execution times required for requested operations. It is, however, difficult for a multimedia application to determine an appropriate deadline that a thread is scheduled as soon as possible and meets it. If the deadline is too soon, the scheduling is not feasible, if it is too far away it may be preempted by another thread.

8 Conclusion

Our extensions to Mach support multimedia applications. These extensions support realtime constraints in the device drivers for continuous media. In particular, we divided multimedia devices into deadline-driven and event-driven types according to their realtime constraints. Standard and pressing realtime threads are scheduled by preemptive deadline scheduling to meet these media-dependent constraints. In addition, this extension helps system designers develop multimedia device drivers efficiently and provide them with a flexible environment.

In the current version, asynchronous event notifications are sometime delayed unexpectedly when other tasks execute long system calls such as *fork*(). We could reduce those event dispatch latencies by modifying the code of these system calls so that the kernel can switch to a realtime thread while executing the code. However, the micro kernel and the Unix Server should give us a solution.

The current version of extended Mach runs on an i386 based personal computer(the FM TOWNS) which supports the following multimedia facilities: MIDI, PCM/FM sound source chips, CD player and video. We implemented user-mode device drivers for MIDI and PCM/FM sound source chips. The system designers reported they were able to debug and develop the device drivers in a more favorable environment than in those of traditional 4.3BSD/Mach. At the time of this writing (March, 1991), we are moving to the micro kernel, or Mach 3.0 based environment.

9 Acknowledgments

We wish to thank the computer music group for helping to implement "Neuro Musician" on our system. We also wish to thank the hypermedia authoring group.

References

[1] Masako Nishijima and Yuji Kijima, "Learning on Sense of Rhythm with Neural Network — The Neuro-Drummer —", in *Proceedings of the First International Conference on Music Perception and Cognition*, October, 1989.

[2] M. J. Accetta, W. Baron, R. V. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young, "Mach: A New Kernel Foundation for UNIX Development", in *Proceedings of the Summer USENIX Conference*, July, 1986.

[3] IEEE, "Realtime Extension for Portable Operating Systems", P1003.4/Draft6, February, 1989.

[4] John A. Stankovic and Krithi Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Operating Systems", *ACM Operating Systems Review*, Vol.23, No.3, July, 1989.

[5] D. Golub, R. Dean, A. Forin, and R. Rashid, "Unix as an Application Program", in *Proceedings of the Summer USENIX Conference*, June, 1990.

[6] J. S. Sventek, "An Architecture supporting Multi-media Integration", *IEEE Computer Society Office Automation Symposium*, Gaithersburg, MD, April, 1987.

[7] David L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", *IEEE Computer*, Vol.23, No.5, 1990.

[8] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao, "Real-Time Mach: Towards a Predictable Real-Time System", in *Proceedings of USENIX Mach Workshop*, October, 1990.

[9] Eric C. Cooper and Richard P. Draves. "C Threads", Technical Report, Computer Science Department, Carnegie Mellon University, CMU-CS-88-154, March, 1987.

[10] C. L. Lui and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM*, Vol.20, No.1, 1973.

[11] Frank W. Miller, "Predictive Deadline Multi-Processing", *ACM Operating Systems Review*, Vol.24, No.4, October, 1990.

[12] Janet I. Egan and Thomas J. Teixeria, "Writing a UNIX Device Driver", *Jhon Wiley & Sons*, 1988.

[13] Tim Thompson, "Keynote - A Language and Extensible Graphic Editor for Music", in *Proceedings of the Winter 1990 USENIX Conference*, January, 1990.

[14] Wei-Kuan Shih, Jane W. S. Liu, Jen-Yao Chung, and Donald W. Gillies, "Scheduling Tasks with Ready Times and Deadline to Minimize Average Error", *ACM Operating Systems Review*, Vol.23, No.3, July, 1989.

[15] James D. Mooney, "Strategies for Supporting Application Portability", *IEEE Computer*, Vol.23, No.11, 1990.

[16] L. Ludwig, "A Threaded/Flow Approach to Reconfigurable Distributed Systems and Service Primitives Architectures", in *Proceedings of ACM SIGCOMM '87*, Stowe, VT, August, 1987.

Jun Nakajima is currently a researcher in the Human Interface Laboratory of Fujitsu Laboratories. He has been researching on expert systems and operating systems since 1986 after graduating from University of Tokyo in Mathematical Engineering. In this project his interests include realtime scheduling for multimedia processing, micro kernel technology, and distributed computing. He is a member of IPSJ and JSAI.

Masatomo Yazaki is currently a researcher in the Human Interface Laboratory of Fujitsu Laboratories. He has been developing multimedia applications at Fujitsu Laboratories since 1987. His interests include multimedia devices and multimedia applications. He is a member of EIC.

Hitoshi Matsumoto is currently a researcher in the Human Interface Laboratory of Fujitsu Laboratories. He received a B.E. degree in applied physics and an M.E. degree in information engineering from Nagoya University, Japan in 1977 and 1979 respectively. He has been engaged in research and development of operating systems, computer graphics, office information systems and artificial intelligence, especially expert systems and tools. His current research interests include hypermedia authoring systems and operating systems dealing with multimedia/hypermedia.

A Testbed for Managing Digital Video and Audio Storage*

P. Venkat Rangan Walter A. Burkhard
Robert W. Bowdidge Harrick M. Vin John W. Lindwall
Kashun Chan Ingvar A. Aaberg Linda M. Yamamoto Ian G. Harris

U.C.S.D. Multimedia Laboratory
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
venkat@cs.ucsd.edu

Abstract

We have developed a testbed for studying multimedia storage systems. We are using the testbed to experiment with media storage issues such as synchronization, continuous recording and retrieval, and manipulation of large files. We define *strand* and *rope* abstractions that provide a paradigm for accessing multiple media. The software architecture of our system consists of two layers: the device-independent *rope server*, and the device-specific *storage manager*. We have implemented a prototype within a Sun and PC environment. We present our experiences and discuss the performance of the system.

1 Introduction

1.1 Motivation

Recent technological advances will make it feasible to integrate transmission and storage of multimedia data with computing. While real-time digital multimedia transmission will have to wait a few more years for very high capacity networks to become pervasive, the integration of storage of digital multimedia with distributed computing merits immediate attention. There are three important features that distinguish multimedia data from those more commonly manipulated by computers, such as text or executable code:

- *Multiple data streams:*

  Video, audio, and other components of a multimedia object originate at different sources and are routed to different destinations. Whereas storing these media streams together entails processing overhead during storage and retrieval, storing them separately requires the file system to explicitly maintain temporal relationships among the streams.

- *Continuous recording and retrieval of data streams:*

  Recording and retrieval of motion video and audio are continuous operations. The file system must organize the multimedia data on disk so that real-time storage and retrieval are guaranteed.

- *Large data size:*

  Video and audio data have very large storage space requirements. For example, one second of NTSC-quality video requires 7 Mbytes of storage, and one second of CD-quality audio requires nearly 200 Kbytes of storage. If the storage system is to act as a basis for supporting media services such as document editing, mail, etc., it must provide time and space efficient mechanisms for manipulating and sharing stored data.

*This work was supported by the NCR Corporation, UVC Corporation, IBM Corporation, Xerox Corporation, the University of California MICRO program, and the National Science Foundation.

A testbed multimedia storage system, which provides a vehicle for experimenting with policies to address the above three requirements, is the subject of this paper.

1.2 Related Research

There are a number of ongoing projects that are investigating the integration of still images and/or audio data with text [2,5,9,11,12,13]. The extension of Etherphone audio storage to video [10] as well as the work by Davenport [4] have tried to store motion video in analog form. There has been very little work on storage systems for digital motion video. The Cambridge Pandora project [6] and Matsushita's Real Time File System [8] have begun investigating low level storage mechanisms for digital video. Both of these systems use special purpose hardware; our approach is to explore the possibilities for using off-the-shelf components.

1.3 Our Contributions

We have implemented a testbed system, with the main emphasis on digital video storage and management. The catalyst for our research has been the availability of PC-based real-time digitization and compression hardware. The testbed provides support for real-time recording and retrieval. We have extended the rope abstraction of the Etherphone system [13], which is a paradigm for accessing audio data, to support multiple media. Our rope abstraction allows manipulation and editing of multimedia data without the overhead of copying large amounts of data. We have also developed a policy to structure multimedia data on the disk in a way that permits random as well as concurrent access.

In the next section of the paper, we describe the hardware environment in our laboratory. In Section 3, we present the software architecture of our testbed, and in Section 4, we discuss some performance measurements. Finally, Section 5 concludes the paper.

2 Laboratory Environment

The hardware environment of our Multimedia Laboratory consists of a number of multimedia stations, each consisting of a Sun SPARCstation, a PC-AT, a video camera, and a video monitor (see Figure 1). The SPARCstations and PC-ATs are connected via Ethernets.

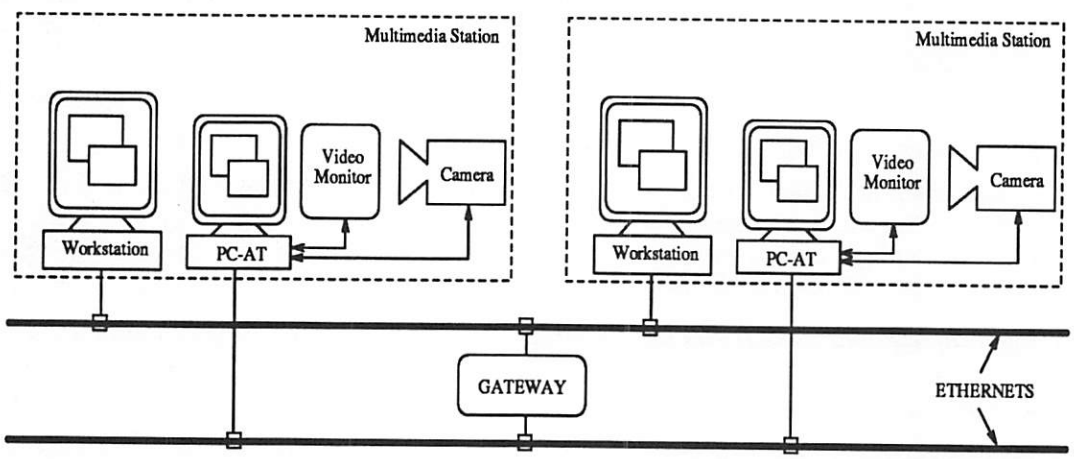


Figure 1: System Configuration

The PC-AT contains video and audio processing hardware produced by the UVC Corporation [7]. The audio hardware digitizes audio signals at 8 Kbytes/sec, and the video hardware captures and displays video at rates up to NTSC broadcast with a resolution of 480x200 pixels (with 12 bits of color information per pixel). Each raw frame is 144 Kbytes in size, but the video hardware uses a run-length encoding technique to limit the size of the encoded frame to an average of 32 Kbytes and a maximum of 64 Kbytes. The

video data is stored on the local disk of the PC-AT, and is displayed on an attached video monitor. The operation of the PC-AT is controlled from the SPARCstation. Communication between the SPARCstation and the PC-AT is accomplished using a TCP/IP socket library. This hardware environment has influenced some of our design decisions.

3 Storage System Architecture

3.1 Preliminary Terminology

Multimedia data includes information in various forms: audio, video, textual, olfactory, thermal, tactile, etc. We anticipate multimedia systems with these varieties of data in the future, but in this paper, we will focus on video and audio. The basic unit of motion video is a *frame*, which contains the data associated with a single video image. The basic unit of audio is a *sample*, which is the amplitude of the analog audio signal at a given instant. A *strand* is a sequence of continuously recorded video frames, audio samples, or both. A *multimedia rope* is a collection of strands together with synchronization information.

3.2 Software Architecture

We designed the software architecture of our system by dividing the storage system into two layers – one responsible for manipulating multimedia ropes (named the *rope server*) and the other responsible for manipulating strands (named the *storage manager*). The rope server communicates with applications and allows the manipulation of ropes. It also communicates with the storage manager to record and play back multimedia strands. The storage manager is responsible for placing strands on disk in order to ensure continuous recording and retrieval.

The functionality of both of these layers was decided by considering the three distinguishing features of multimedia storage. The rope server manipulates references to strands rather than the data itself to limit the need to copy video and audio data. It also is responsible for maintaining synchronization information about strands – information stored as part of the rope. The storage manager handles continuous recording and retrieval of the strands. Since the storage manager is responsible for device-dependent functions (such as deciding how to place the data on disk), these two layers also separate the device-dependent functions of the storage system from the device independent functions.

3.2.1 Interface Provided by the Rope Server

The rope server implements the rope abstraction. Applications can use the methods in Table 1 to manipulate ropes. The *rope access* methods are modelled after the UNIX file access routines. The motivation for this is (1) to keep a UNIX-like paradigm for operating on objects (that is, *open object*, *manipulate object*, *close object*) and (2) to allow rope locking and read-ahead optimizations within the system. The two *editing* methods can perform all needed rearranging, copying, and deleting of data. The *real-time access* methods are non-blocking, returning immediately with an instanceID. Applications use instanceIDs to refer to these requests in subsequent operations (such as stopping an earlier record or playback request). To keep the application informed about the state of the play or record operation, the rope server sends messages at intervals to tell the application of the current location being played or recorded within the rope. Messages are sent at the beginning and end of a play or record request, when a disk error occurs, and at intervals during the playback of the rope. We call these messages "callbacks" since the rope server returns information about the play or record function call after the call has returned. All methods are time-based, thereby obtaining a frame-rate independent interface.

3.2.2 Interface Provided by the Storage Manager

The storage manager implements strands. The interface between the storage manager and the rope server includes four methods for manipulating strands (see Table 2). PlayStrandSequence takes a sequence of

```
[strand reference, start time, end time]
```

ROPE ACCESS:	rope descriptor	= OpenRope(rope name, access rights)
	status	= CloseRope(rope descriptor)
	status	= DeleteRope(rope name)
EDITING:	status	= InsertInterval(destination rope desc.,insert point, source rope desc., time interval of rope)
	status	= DeleteInterval(rope descriptor, time interval of rope)
UTILITY:	time	= GetLengthOfRope(rope descriptor)
	rope information	= GetRopeInfo(rope descriptor)
	status	= SetRopeInfo(rope descriptor, rope information)
REAL-TIME ACCESS:	instanceID	= PlayRope(rope descriptor, time interval of rope, callback info)
	instanceID	= RecordRope(rope descriptor,position within rope to insert recorded segment, callback info)
	status	= StopRope(instanceID)
	status	= PauseRope(instanceID)
	status	= ResumeRope(instanceID)

Table 1: Interface between the Application and rope server

objects (called strand intervals) and displays the given time interval of each strand in sequence. RecordStrand creates a new strand and records video and audio data either for a given duration or until StopStrand is called. StopStrand terminates a previous PlayStrandSequence or RecordStrand instance. DeleteStrand is used to remove an entire strand object from storage.

instanceID	= PlayStrandSequence(sequence of [strand reference,start time, end time])
instanceID	= RecordStrand (duration)
status	= StopStrand (instanceID)
status	= DeleteStrand (strand reference)

Table 2: Functions for the rope server to access the storage manager

3.3 Implementation

The rope server of our prototype storage system runs on the SPARCstation and the storage manager runs on the PC-AT. Applications are compiled with a multimedia rope server library which uses remote procedure calls to contact the rope server. The rope server and storage manager communicate using TCP/IP. The ropes are stored in the UNIX file system as text files. The strands are stored on the PC-AT's local disk.

The motivation behind this division of functionality is that it allows the rope server (the device-independent part of the storage system) to be decoupled from the storage manager (which uses device-specific algorithms). Since the video capture and display hardware was only available on the DOS platform, and since we were forced to store video strands on the PC-AT due to network speeds, we decided to locate the storage manager on the PC-AT. The rope server was then implemented on the SPARCstation for portability.

3.3.1 Rope Server

The rope server stores each rope as a sequence of strand intervals. To facilitate copy-free editing, operations on ropes manipulate only these strand intervals when performing an action. For example, to copy a section of a rope, InsertInterval finds the set of strand intervals defining a range of time and copies these intervals into the destination rope at the proper insertion point. To play a section of a rope, PlayRope finds the sequence of intervals that defines the rope to be played. These strand intervals can be sent to the storage manager which then plays the proper part of each strand in sequence. Strand intervals also allow strands to be shared by multiple ropes. A reference count mechanism (similar to that described in [13]) is used to delete strands no longer referenced in any rope.

As a rope is played or recorded, the storage manager sends callback messages to the rope server at the beginning and end of the sequence, at the beginning and end of each strand interval in the sequence, and when a disk error occurs. The callback messages are also sent by the rope server to the application in order to keep it informed about the state of an action.

3.3.2 Storage Manager

In our prototype storage system, a strand consists of a sequence of media blocks, each block containing a single video frame and associated audio. Performance requirements of recording and retrieval operations demand that the storage manager ensure access to consecutive blocks in real-time. It should also be possible to start playback from an arbitrary frame of a strand with minimal delay. The storage manager meets these requirements by using a two-level index structure similar to the UNIX file system's use of indirect blocks. This structure allows for fast access to any block within the strand. The two level index structure contains a header block, one or more index blocks, and one or more secondary blocks. An index block contains references to a sequence of contiguous frames, a secondary block contains references to a sequence of index blocks, and the header block of the rope contains references to all secondary blocks. This allows us to quickly start playing video from any location in the strand with a minimal number of disk accesses.

In the current implementation, each index and secondary block is 1024 bytes long, and each reference to a data block requires 6 bytes, so each index block contains references to (1024/6=170) data blocks. Similarly, each secondary block contains references to 170 index blocks. We are experimenting with optimizing all the parameters for the storage system architecture.

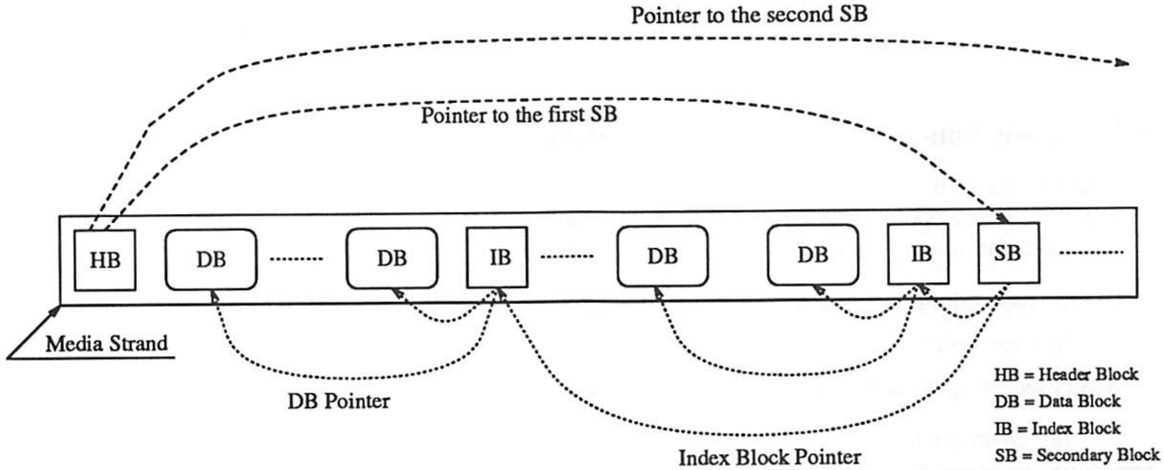


Figure 2: Structure of strands on disk

4 Experience and Performance Evaluation

4.1 Rope Editor

The first application that we have developed using the storage system is a multimedia rope editor. The rope editor allows a user to create new ropes, display existing ropes, copy an interval of one rope to another, and delete intervals of ropes. A graphical interface built using the XView library provides an attractive and easy-to-use interface (see Figure 3). The ease of implementation of the rope editor suggests the appropriateness of the interfaces to the system.

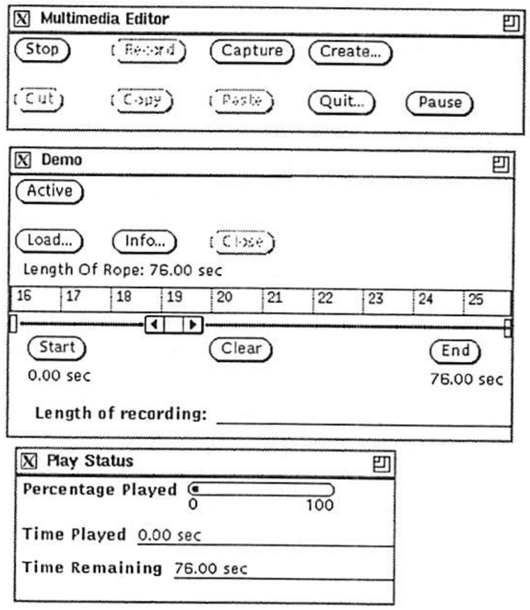


Figure 3: User interface for the multimedia rope editor

4.2 Performance of Storage System and Applications

To evaluate the testbed, we analyze the performance of the PLAY, RECORD, and STOP operations of the rope editor. These operations exercise all layers of the storage system.[1]

Consider the play operation. On clicking PLAY, the following actions are taken:

1. The rope editor makes a remote procedure call (PlayRope) to the rope server.
2. The rope server determines the strands to be played, and sends the list to the storage manager.
3. On receiving a request, the storage manager performs the following operations:

    (a) Open a strand.

[1] In all our experiments, the video recording rate was set to 15 frames per second, and the frame size was chosen to be 384x148 pixels.

(b) Read the header block, and the first index and secondary blocks.

(c) Access audio and video blocks using the index structure and send them to the display hardware. Read successive index and secondary blocks as needed.

(d) Close the strand on completion of its retrieval. If all the strands necessary for playback have been displayed, then terminate the playback. Otherwise, repeat (3) for the remaining strands.

Clicking RECORD causes a similar set of actions to be taken.

1. The rope editor makes a RPC (RecordRope) to the rope server.

2. The rope server requests a new strand to be recorded by the storage manager.

3. The storage manager opens a new strand for writing.

4. Video and audio are written to disk.

5. The rope server inserts the new strand into the proper rope.

When STOP is clicked, the rope editor makes an RPC to the rope server. The rope server propagates the stop request to the storage manager, which terminates the operation (recording or playback) in progress.

Table 3 shows measurements of the delay between (1) clicking PLAY and the start of playback, and (2) clicking STOP and the termination of playback as a function of the length of a media strand.

	PLAY		STOP	
Duration of media strand in seconds	mean delay to start playback	95% confidence interval of mean	mean time to stop display	95% confidence interval of mean
5 s	183 ms	181 - 186 ms	61 ms	58 - 64 ms
30 s	216 ms	211 - 220 ms	64 ms	60 - 68 ms
300 s	427 ms	416 - 439 ms	70 ms	66 - 75 ms

Table 3: Delay measurements of the PLAY and STOP operations in the rope editor

duration of media strand in seconds	mean time to open (reading)	95% confidence interval of mean
5 s	112 ms	111 ms - 113ms
30 s	129 ms	128 ms - 130 ms
300 s	309 ms	308 ms - 310 ms

Table 4: Measurements of time required by PC-AT disk access library to open strands of various durations.

The media strands are opened and manipulated using a disk access library (provided by the UVC Corp.) optimized for accessing video at a high rate. The optimizations include prefetching and preallocating blocks on the disk while opening a strand. Hence, the time to open a strand depends on its size (see Table 4). The measurements in Table 3 and Table 4 illustrate that the overhead of opening a strand contributes significantly to the delay in initiating a playback operation. The difference between the values in Table 3 and Table 4 represents the overhead of the rest of the system, which varies from 70 to 120 ms. This variation can be attributed to the difference in seek time: longer strands have greater separation between the header block and the first index and secondary blocks.

We find the delay between clicking RECORD and the start of recording is 1.163 s. Most of this time is spent creating a new strand and preallocating space for it to ensure that video can be written to disk

quickly. This preallocation performed by the disk access library causes the time to create a file to be 1.057 s. Hence, the overhead of the rest of the system is about 106 ms. Terminating a RECORD operation (using STOP) requires the same amount of time as the termination of a PLAY operation (71 ms on the average).

Since the disk access library only allows one strand to be open at a time, playing a rope consisting of a sequence of strands requires one strand to be closed before the next can be opened. The overhead of switching causes a pause during the playback process at strand boundaries. The overhead for switching is summarized in Table 5. Once again, much of the overhead can be attributed to the time required to open a strand.

Duration of Strands	Switching Overhead	95% confidence interval of mean
5 s	168 ms	164 - 171 ms
30 s	210 ms	208 - 213 ms
300 s	384 ms	374 - 395 ms

Table 5: Overhead of switching between strands

Table 6 shows average execution times for various rope manipulation operations supported by the storage system. Note that the execution times of almost all the rope manipulation operations are very small. Functions such as InsertRegion and GetRopeInfo operate on data structures residing in memory, and so operate extremely quickly. Other functions such as OpenRope and CloseRope take longer to execute because they must open and process ropes (represented by UNIX files). DeleteRope requires a significantly larger time to complete since its execution involves processing by the storage manager.

Rope Manipulation Function	Mean Service Time	95% confidence interval of mean
OpenRope		
1 strand rope	13 ms	12.8 - 13.6 ms
6 strand rope	15.7 ms	14.0 - 17.5 ms
CloseRope	17.5 ms	16.5 - 18.7 ms
InsertRegion	2.06 ms	2.04 - 2.08 ms
DeleteRegion	2.12 ms	2.10 - 2.14 ms
GetRopeInfo	2.4 ms	2.3 - 2.5 ms
SetRopeInfo	2.8 ms	2.5 - 3.0 ms
GetLengthOfRope	2.5 ms	2.0 - 2.9 ms
DeleteRope	180 ms	178 - 183 ms

Table 6: Execution times for rope manipulation operations

To summarize, the performance of our storage system is quite satisfactory, and the response time of the rope editor application is low enough for it to be a useful tool. Some of the performance figures can be significantly improved by developing a disk access library optimized for our strand format. We are also experimenting with various storage placement algorithms and their effect on system performance.

5 Conclusion

The prototype storage system that we have presented serves as a testbed for studying issues of multimedia file storage and management. We have presented the software architecture of our system, and described a technique for structuring video files on disk that allows for real-time recording and retrieval. The multimedia rope abstraction eliminates copying of raw data on disk during editing operations. The callback mechanism

informs the application about the state of the recording and retrieval requests. The performance of our system has been very satisfactory.

We are using the prototype storage system to experiment with various policies for (1) determining the granularity for storing audio and video data on disk, and (2) resource allocation to permit multiple concurrent accesses to a multimedia storage system. We are currently developing protocols for accessing stored media data across the network. We are also enhancing the storage system with mechanisms to independently manipulate individual media streams. We believe that the experience gained from building this prototype will prove valuable in the design of storage systems for the multimedia environments of the future.

References

[1] David P. Anderson, Shin-Yuan Tzou, Robert Wahbe, Ramesh Govindan, Martin Andrews, "Support for Continuous Media in the DASH System", Proceedings of the 10th International Conference on Distributed Computing Systems, Paris, May 28 – June 1, 1990, pp. 54-60.

[2] Barry Arons, Carl Binding, Keith Lantz, Chris Schmandt, "The VOX Audio Server", Multimedia '89: 2nd IEEE COMSOC International Multimedia Communications Workshop, Ottawa, Ontario, April 20-23, 1989.

[3] S. Casner, K. Seo, W. Edmond, C. Topolcic, "N-Way Conferencing with Packet Video", Third International Workshop on Packet Video, Morristown, NJ, March 22-23, 1990.

[4] G. Davenport, "Video File Systems", private communication, MIT Media Lab, Cambridge MA, February 1990.

[5] Simon Gibbs, Dennis Tsichritzis, Akis Fitas, Dimitri Konstantas, Yiannis Yeorgaroudakis, 1987, "Muse: A Multimedia Filing System", IEEE Software, 4(3):4-15 (March 1987).

[6] Andy Hopper, 'Pandora – an experimental system for multimedia applications", Operating Systems Review 24(2):19-34 (April 1990).

[7] Milt Leonard, "Compression Chip Handles Real-Time Video and Audio", Electronic Design, December 13, 1990, pp. 43-48.

[8] Yoshihiro Mori. "Multimedia Real-Time File System", Matsushita Electric Industrial Co., presentation at U.C. Berkeley, February 21, 1990.

[9] B.C. Ooi, A.D. Narasimhalu, K.Y. Wang, I.F. Chang. "Design of a Multi-Media File Server using Optical Disks for Office Applications". Proceedings of the IEEE Computer Society Office Automation Symposium, Gaithersburg, MD, April 1987.

[10] P. Venkat Rangan and Daniel C. Swinehart. "Video Conferencing and File Storage in the Etherphone System", submitted to IEEE JSAC, December 1990

[11] Sun Microsystems, Inc., August 1989. "Multi-Media File System Overview".

[12] Robert H. Thomas, Harry C. Forsdick, Terrence R. Crowley, Richard W. Schaaf, Raymond S. Tomlinson, Virginia M. Travers. "Diamond: A Multimedia Message System Built on a Distributed Architecture", IEEE Computer, 16(12):65-78 (December 1985).

[13] Douglas B. Terry and Daniel C. Swinehart. "Managing Stored Voice in the Etherphone System", ACM Transactions on Computer Systems, 6(1):3-27 (February 1988)

P. Venkat Rangan received a B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, India, in 1984, and a Ph.D. degree in computer science from the University of California, Berkeley in 1988. Since 1989, he has been an Assistant Professor of Computer Science and Engineering at the University of California, San Diego, where he co-directs the Multimedia Laboratory. He is a member of IEEE and ACM.

Walter A. Burkhard is a Professor of Computer Science and Engineering with research interests in mass storage systems, fault tolerant computation, and distributed computation. He has served as Department Chairman, and is co-directing the Multimedia Laboratory. He is a senior member of IEEE and a member of ACM. He received his Ph.D. in electrical engineering and computer science from the University of California, Berkeley in 1969.

Robert W. Bowdidge is a graduate student in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include file systems, distributed computing, and computer graphics. He is currently a student member of USENIX and ACM. Mr. Bowdidge received a B.A. in computer science from the University of California, Berkeley in 1989, and an M.S. in computer science from the University of California, San Diego in 1991.

Harrick M. Vin is a doctoral candidate in the Department of Computer Science and Engineering at the University of California, San Diego. He received his B.Tech. in computer science and engineering from the Indian Institute of Technology, Bombay in 1987 and his M.S. in computer science from Colorado State University in 1988. His research interests include multimedia computer systems, ultra-high speed networking, parallel architectures, and highly optimizing compilers. He is a recipient of an IBM Doctoral Fellowship, and a student member of IEEE and ACM.

John W. Lindwall is a Systems Specialist for Quintessential Solutions, Inc. His current interests include scientific visualization, virtual reality, and image compression. Mr. Lindwall received a B.S. in computer science from California State University at San Diego, and an M.S. in computer science from the University of California, San Diego.

Kashun Chan is currently an undergraduate in the Department of Electrical and Computer Engineering at the University of California, San Diego. His research interests include multimedia computer systems, ultra-high speed networking, and queueing systems. He has been involved in research in the Multimedia Laboratory since the summer of 1990.

Ingvar A. Aaberg is currently an undergraduate in the Department of Computer Science and Engineering at the University of California, San Diego. He has been a research assistant in the Multimedia Laboratory since the fall of 1990. Ingvar will start graduate school at U.C.S.D. in the fall. His interests include synchronization issues in multimedia systems, distributed systems and neural networks.

Linda M. Yamamoto is completing an M.S. in computer science at the University of California, San Diego. Her interests in the multimedia field include its interaction with human cognition. Currently, she works for Hughes Network Systems in San Diego. Ms. Yamamoto received a B.S. in electrical engineering and computer science from the University of California, Berkeley in 1988.

Ian G. Harris is a graduate student in the Computer Science and Engineering Department at the University of California, San Diego. His current interests are multimedia systems and human interfaces. His past experience includes one year at the Media Lab at the Massachusetts Institute of Technology, where he completed his undergraduate thesis on musical cognition. Mr. Harris received a B.S. in computer science from the Massachusetts Institute of Technology in 1990.

The Architecture of the IRCAM Musical Workstation

Eric Lindemann
Miller Puckette
Eric Viara
Maurizio De Cecco
Francois Dechelle
Bennett Smith

Institut de Recherche et Coordination of Acoustique/Musique
31, rue Saint-Merri
75004 Paris, France
elind@ircam.fr

Note: This paper includes extracts from a number of articles accepted for publication in 'Computer Music Journal' 15(3) Fall, 1991, MIT Press.

The IRCAM Musical Workstation (IMW) is designed to facilitate experimentation with real-time signal processing and event processing. The main focus is on interactive musical composition and performance algorithms. The hardware architecture of the IMW is based around a general-purpose multi-processor providing sufficient number crunching power for real-time musical signal processing and event processing.

A real-time operating system has been written for this multi-processor as well as a toolbox which provides support for distributed real-time signal processing and event processing applications. The toolbox also defines an object system to support interactive CAD-like applications. Musical support in the form of MIDI, digital audio I/O, and musically oriented event scheduling has been provided.

The multi-processor consists of a number of processor boards--each with two Intel i860 processors [Intel 1989]--that plug into a host computer--the NeXT machine. The host is used for its graphic interface and file system. The i860 processors handle all real-time event processing and signal processing.

A number of distributed applications have been, or are being, written for the system. User interfaces run on the host and communicate with tasks on the multi-processor. These include: Animal [Lindemann et al. 1991]--an object oriented data definition and manipulation environment, MAX [Puckette 1991] --a graphic programming language, IMW Signal Editor [Eckel 1990]--a tool for manipulating sampled signals in time and frequency domains, and the IMW Universal Recorder [Smith 1990]--a multitrack real-time data recorder for spooling event and signal streams to disk.

This article will provide a general overview of the IMW system with attention given to the hardware architecture, the real-time operating system and toolbox, and the Animal graphic programming tools.

Musical Signal Processing Architectures

Previous systems for real-time musical signal processing have been divided into several subsystems: a host computer--Apple Macintosh, IBM PC, or Unix Workstation--serving as a general development environment, graphic user interface, and file server; a real-time control system used for general event handling--MIDI controllers, voice allocation; and a real-time signal processing system. The control section typically includes a number of microprocessors--Motorola 680xx, Intel 80x86--running real-time kernels. In some low end systems the task of real-time event management is carried out by the host computer. This tends to work reasonably well with PCs and much less well with workstations. The signal processor can take a number of forms: custom non-programmable designs highly optimized for particular algorithms--FFTs, table look-up oscillators; moderately programmable designs optimized for a certain class of algorithms; more fully programmable designs, either custom architectures or ones based around off the shelf DSP processors. Examples of programmable systems which are or have been used for musical production are the IRCAM 4X machine [Di Guigno et al. 1986], the WaveFrame AudioFrame [Lindemann 1987, and the Lucas Film SoundDroid machine [Moorer 1982].

As a vehicles for experimentation, the multilevel architecture described above can quickly become unwieldy. There is generally a separate development environment for each type of processor--host, control, signal processing. An application program must generally be written for each processor and these programs must be made to communicate reliably. The signal processing routines are either microcoded or written in some extended assembler with parallel instruction capabilities. In addition the signal processing routines may need to be written for integer processors, complicating the task of implementing complex algorithms.

This process tends to be discouraging to researchers. Most research at IRCAM has, in fact, been based on simulation using standard non-real-time systems--workstations and VAXes. Only certain well suited algorithms have had real-time implementations. These types of systems, with several flavors of processors, also tend to be rather costly and poorly adapted to time-sharing. As a result, they are scare resources, ill-suited to casual experimentation.

Experience with the types of systems described above led to a statement of goals for a new real-time musical signal processing architecture:

1. Reduction of the number of types of processors and development environments in the system.
2. Minimization of specialized low-level programming requirements
3. Use of floating-point numbers throughout.
4. Development of a rich set of rapid prototyping tools including graphical programming languages and data visualization environments.
5. Cost reduction.

The IMW Calculation Engine

1989 saw the introduction of a general-purpose extended RISC processor, the Intel i860, capable of very high-speed real-time floating-point number crunching. Its 25nsec. cycle time, its ability to perform a floating-point add, a floating-point multiply, and an integer operation every cycle, its integrated caches and MMU made it the ideal choice for the IMW. The i860 made possible the unification of the real-time control and signal processing environments, turning a three level system--host, real-time control, DSP-- into a two level system--host, real-time--while at the same time holding out the promise of a high level environment for developing signal processing algorithms.

The calculation engine is implemented as a plug-in coprocessor board for the NeXT computer. A block diagram of the board is shown in Fig. 1.

Each board has two i860 processors with up to 32 MBytes of local dynamic RAM. The processors communicate by reading or writing each other's RAM. Interconnection between processors on the same board is through a crossbar switch.

The 56001 serves a I/O processor also serves as direct memory access (DMA) controller for burst transfers between itself and the local memory of i860 processors, and between the memory of i860 processors on different boards.

The four slot NextBus permits three dual-processor i860 boards to be plugged into the same machine alongside the host CPU board, which occupies one slot. The NextBus is an extended version of the Texas Instruments NuBus with a double-speed burst mode which permits 40 nsec 32-bit transfers. This gives a theoretical maximum transfer rate of 100 MBytes/sec between boards plugged into the NextBus. The practical rate after arbitration is somewhat less than half of that.

CPOS/FTS--Real Time Oprating System and Software Toolbox

An operating system, CPOS, and a real-time distributed program, FTS, form the software platform for real-time musical programming on the IMW.. Graphical applications, such as Animal, the sound editor, the and MAX, use CPOS and FTS to create real-time synthesis or control objects and communicate with them.

CPOS (Co-Processor Operating System) supports general purpose, scientific and real-time applications for Digital Signal Processing, taking advantage of IMW's specific architectural features. CPOS is mainly composed of a kernel running on the CPs and a driver on the NeXT host machine for host/CP communication.

For general-purpose applications, CPOS offers a set of system calls based on a subset of standard UNIX. The system calls include filesystem access (open(), read()...) and memory/task management

(sbrk(), fork()...). Using a C or FORTRAN program development environment (compiler, assembler, loader, libraries) for the i860, any portable application can run in the CPOS environment. The file system calls are done by a NeXT server via the driver; there is no local file system.

For memory and process management, CPOS offers a set of system calls adapted to the (multiprocessors and local memory) architecture. We can chose which CPU to run a process on and specify a physical zone for memory allocation. For communication between processes, CPOS offers an Inter-Process Communication package based on formatted messages and a package for shared memory management between processes.

The response time in a typical operating system to an external interrupt is about 1 millisecond (100 to 200 microseconds on a i860). In real-time audio processing where the computation of N samples cannot be more than N/F0 = N*22 microseconds (for 44KHz), this would clearly not be fast enough. CPOS response time to a an external interrupt is about 30 to 40 microseconds, consisting only of context saving and restoring.

CPOS introduces a real-time mode that can give a certain task priority over all others (many operating systems allow that much), but also forbids the processing of all external interrupts except the ADC/DAC clock - used only for time reference - and those set by processes running in super-preemptive mode on other processors. Testing for the origin of an external interrupt and processing a clock interrupt take only 0.6 microseconds each.

FTS ("Faster Than Sound") consists of an object system together with a collection of classes, defined in C, which can be instantiated from a CP or from the host. A protocol is defined for passing messages between objects in FTS, either locally or between CPs. FTS also supplies support for MIDI and sound I/O and reschedulable delayed callbacks.

The FTS object system was designed specifically for real-time music applications. In many respects it is much simpler than most object systems, but it provides a combination of services unique among c-language message systems, that is needed in our context. Its most unusual point, among C message systems, is that messages are objects which can be copied and stored, whose arguments are typed. The FTS message system can check the argument types of the message against the types taken by the receiving object's method for the message. The typing of message arguments also facilitates transmission across machine boundaries; byte swapping is necessary when passing messages between the NeXT host and a CP. The FTS object system is also unusual (among C object systems) in that one can dynamically install new classes, and (with some care) change a class's instance data structure and/or methods.

An FTS message consists of a selector, which is a pointer to a symbol, and zero or more typed arguments. The fundamental operation defined for a message is to pass it to an object.. The caller assembles the arguments for the method into a contiguous data structure and calls FTS's message routine. This routine looks up the receiving object's entry for the message, in a table that the first slot of the receiving object's data structure points to. This entry gives a pointer to the method and an argument type template. The message passing routine checks the argument types and calls the method.

The FTS task's inputs all appear as time-tagged queues. Except for the serial input queue and the timeout queue, they all share the same structure. This general queue structure treats messages and sound differently. In each queue slot (the contents of a queue for a specific tick) there is a subqueue of messages, of undetermined length, and a predetermined number of signal buffers. In the case of sound input, the message part of the queue is empty.

For each tick, the FTS carries out its (message and DSP) duty cycle as follows. The task empties out, in sequence, the message contents of each of its queue slots for that tick, passing each message to its destination. (In the case of the serial port and timeout queues, this is not an FTS message pass but a prearranged function call.) Before it processes the tick, it waits until all the queue slots for that tick have been filled, i.e. that the task or device which fills the queue has promised that no more information will be sent for the given queue slot. The queue slot associated with sound input is processed last; instead of looking in the (empty) message portion of the slot, FTS runs the DSP duty cycle for that tick, which handles sound production.

MAX

MAX (Puckette 1988; Opcode 1990) is a graphical programming environment for developing real-time musical applications. First written for the Apple MacIntosh computer, it has been ported to the NeXT computer as a part of the IRCAM Music Workstation ("IMW") project (Lindemann 1991a). From its earliest conception, MAX was intended as a unified environment for describing both control and signal flow. Historically it has developed as a MIDI (i.e., control) program primarily because the 4X (Favreau 1986), IRCAM's earlier signal processing engine, could only communicate with the MacIntosh over a MIDI (serial) line.

The main purpose for making electronic music production run in real time is so that a musician can exercise some sort of live control over the music. The problem of defining that control is a much harder one than that of defining the signal processing network which ultimately will generate the samples. The sample generation problem has historically been considered "hard" simply because of its stringent computational requirements. Today, a real-time programmable audio synthesis and processing engine can be bought at a price that researchers, and even some musicians, can pay. It is therefore not surprising that many systems are now being proposed for graphical signal network editing. But the control problem, that of making the signal network respond in an instrument-like way to live human control, is not made appreciably easier by the availability of faster and faster hardware. Today, the challenge for a signal processing network editor is to open itself up to a wide range of control possibilities.

To take complete control of all the possibilities of some kind of signal processing "patch," or network, it may be necessary to specify independently where all the control is coming from: the basic pitch and tempo material, timbral changes, pitch articulation, whatever. These should be controllable physically, sequentially, or algorithmically; if algorithmically, the inputs to the algorithm should themselves be controllable in any way. The more a given situation relies on unusual synthesis methods or input devices, the more acutely we need to be able to specify exactly what will control what, and how.

The fundamental concept in MAX is the patch. A patch is a collection of boxes connected by lines. The boxes represent objects which wait for messages to be passed to them, at which time they may respond by passing messages to other boxes. Boxes may have inlets and outlets, which appear as dark rectangles on top of and on bottom of the boxes. The line segments connect outlets to inlets; any message the source object passes to its outlet is passed on to all the inlets connected to it. Patches may be nested as deeply as desired.

The messages that are passed down the lines consist of an ordered list of atoms, each of which may be a number (fixed or floating point) or a symbol. Any message that can be passed has a printable equivalent. Messages frequently consist of a single number, or of the symbol "bang," which is used conventionally to denote an event which has no parameters.

In addition to sending and receiving messages (usually via the inlet/outlet mechanism), objects in MAX may access the clock or MIDI I/O. The clock is accessed via a simple callback mechanism. An object may allocate any number of desired "virtual clock" objects. Each virtual clock may be set to call the client object back at a given time; the callback time of a virtual clock may be changed at will or the callback may be cancelled. There is only one priority. An object wishing to receive incoming MIDI messages inserts itself on the appropriate MIDI callback list. MIDI output is spooled by calling a library function.

The scheduler for MAX is provided by the FTS system (Puckette 1991), which also provides a DSP duty cycle clock which is used for the signal processing extension, and an interprocessor messaging facility.

MAX is not a dataflow language; the boxes which make up a patch usually contain some local state. Dataflow's independence of the order in which the inputs to an operation become available cannot be achieved here. On the other hand, MAX's object-oriented approach is much more appropriate for systems which must respond to external requests for action. In this scenario, which is typical of live human/machine interaction, the order in which transactions occur is often significant. A violin should be tuned before playing it, not after, for best results.

Signal processing in MAX is carried out by a collection of "tilde classes" which communicate via inlets and outlets through the message, "signal." As a convention, the names of tilde classes all end in tilde, as in sig~, osc1~, etc.

The tilde objects carry out signal processing tasks on vectors of a fixed size N, typically between 16 and 64. The DSP duty cycle time is set to the sampling rate divided by N. For each DSP duty cycle period,

every running tilde object is called to carry out its duty cycle action. This action may reference signal inputs and/or outputs (which appear as vectors of size N), and/or the tilde object's instance space.

The tilde objects must intercommunicate at setup time in order to determine a calling order, and the addresses of input and output signals, to be used in the DSP duty cycle. This is managed via the "signal" message, which the tilde objects pass among themselves via inlets and outlets. "Signal" is an ordinary MAX message, and no extension of the inlet/outlet mechanism was made to introduce it.

Tilde classes communicate with control objects through their instance data. For example, a two-pole filter, f2p~, is defined which maintains three filter coefficients which are used during the DSP duty cycle. These coefficients may be changed via messages, since the same instance data an f2p~ uses during DSP processing is accessible to it as it handles messages.

A simple patch, shown in Fig. 2, outputs a cosine wave whose frequency is controlled by incoming MIDI note-on messages. The signal-processing part of the patch is defined by the sig~, osc1~, line~, *~, and dac~ objects. The sig~ is a message-to-signal convertor, taking floating-point numbers as input and creating a signal output whose samples are all equal to the most recent value received. The osc1~ takes a frequency and phase offset as signal inputs and outputs a cosine wave (calculated by interpolating table lookup) of unit amplitude. Since its phase offset input is disconnected it is taken as zero. The cosine is then multiplied (via *~) by the output of the line~ breakpoint envelope generator, which may be passed breakpoints as messages of the form (target-value time-in-milliseconds). The dac~ then sends the result to both channels of audio output.

The message box at upper left is used as a button; clicking on it passes the messages "start" to the dac~, "440." to the sig~, and "0.1 50" to the line~. (The objects such as "r freq" act as remote message receivers; "s freq" is a remote message sender.) The chain at lower left takes in MIDI note-on messages, discards those of velocity zero, displays the resulting pitch (ignoring velocity), converts to a frequency and sends it to the "r freq." The two message boxes at top center ramp the amplitude up and down when activated.

In this example, three asynchronous event sources are merged: mouse clicks, incoming MIDI, and the DSP duty cycle. The boundaries between event sources occur at the inlets of the tilde objects, which change their state in ways that is later reflected in their DSP behavior.

Animal

Animal (Animated Language) is a rapid software prototyping tool designed for experimentation with real-time signal processing and event processing systems. Animal is an objected oriented programming environment with the usual notions of class, method, and instance. A class is defined in Animal by designing its graphic representation(s). These representations then serve as interfaces for creating and manipulating networks of "live" instances of classes. Animal provides for persistent storage on disk of instance networks, and for reusable libraries of classes, graphic representations, and instances.

Animal is intended as a tool for building "fine-grained" graphic applications: musical event list editors, synthesizer patch editors, etc.

An Animal class definition specifies a template data structure consisting of primitive objects (float, int, string, etc.), arrays of primitive objects, pointers to other objects which are instances of classes defined in the class table, and lists or sets of pointers to objects. All the objects in an Animal application are instances of classes in the class table. These objects live in core memory on the real-time multiprocessor. The graphic representations of these classes live on the host computer. The methods for classes generated using Animal are defined by the application designer and written in C or C++. All these methods run on the real-time multiprocessor. The Animal application designer writes no code which runs on the host.

With their pointer references, the objects in an Animal application form an evolving object network. Animal maintains a separate "proxy network" on the host computer which is a direct reflection of the object network on the multiprocessor. There may be one or more graphic representations defined for each class in the class table. A graphic representation is said to be bound to the class it represents. A graphic representation maintains slots which are bound to instance variables of the class. There may be more than one slot bound to the same instance variable. Slots, in turn, contain representations of the instance variables they are bound to. These instance variables may be primitive objects--floats, ints, strings--or pointers to complex objects. The representation contained in a slot may be a complex representation of the object pointed to by this instance variable. So, representations are recursive structures with representations

containing slots containing representations ...

The graphic representations interact with their multiprocessor instances through the intermediary of the proxy network. As the instance variables of objects are modified on the multiprocessor, update messages are sent to the corresponding proxy objects in the host network. When a proxy object receives an update message it broadcasts it to all visible representations of that object.

When the user interacts with a representation in a way which is intended to modify some instance variable then a message is sent to the proxy object which in turn sends a message to the multiprocessor object. Update messages are also sent to any other visible representations of the instance.

At any time there may be many windows visible on screen with representations of instances of different classes, of different instances of the same class, or with multiple representations of the same instance.

The archiving of an Animal application is accomplished by archiving the proxy network. The multiprocessor system is sent a message to update all host objects. When this is complete the proxy network is archived along with the class table for the application. The multiprocessor network is not archived directly. While dearchiving the proxy network on the host, messages are sent to the FTS object factory to rebuild the multiprocessor network in the image of the proxy network.

A Sampling Synthesizer Designed in Animal

Figures (3-5) show the interface for a sampling syntheizer application designed using Animal. The interface is hierarchical in nature, appearing as a tree of full winodw representations. At the top level of the hierarchy is the orchestra representation shown in Fig. 3. The orchestra consists of a collection of iconic representations of instances of the instrument class. The relative height of the instrument icons represents the relative overall amplitude of the instrument. Amplitude is an instance variable of the instrument class. By resizing the icon using the mouse this relative amplitude can be adjusted. New instruments can be created by cloning the prototype instrument icon which appears above and to the left of the instrument collection. Double-clicking the mouse on an instrument icon opens a full window representation of the instrument class(Fig 4.).

The instrument window is dominated by a collection of "samplePatch" icon representations which have been mapped onto a rectangular region called the "keymap." The keymap provides a coordinate system with MIDI pitch on the vertical axis and MIDI velocity on the horizontal axis. So, each samplePatch icon covers a region in pitch-velocity space. A sample patch object encapsulates a single sampled sound--a recording of a gong, chime, etc. --and a number of controls associated with that sound. When a note is played on a MIDI keyboard controller, or in general when the instrument object receives a "note-on" message, those sample patches whose mapping covers the point specified by the pitch and velocity of the note-on message will be activated. New samplePatches can be cloned from the prototype found above and to the left of the keymap and placed in the keymap

The Keymap and the collection of instruments in the orchestra window are examples of Animal Ruler-Sets. Rulers measure the position of objects placed within them. Sets represent set instance variables--collections of objects of specified type(s).

Properties associated with representations and slots allow the Animal designer to specify that a particular representation functions as a prototype, i.e. it can be copied but not moved or deleted, and that new instances can only be placed in specified slots, e.g. new samplePatch icons can only be placed inside the Keymap Ruler-Set.

It can be seen that archive properties define the scope of composite Animal objects. This is because all instance creation in Animal is based on cloning existing instances.

Double-clicking on a samplePatch opens the samplePatch window, visible in Fig. 5, with its internal structure consisting of Sound icon, amplitude envelope editor, polePlot of a twoPoleFilter, and centerFrequencyBandwidth indicator for the same twoPoleFilter. The twoPoleFilter coefficients can be adjusted either by dragging on the X representations in the polePlot or by dragging and resizing the light grey rectangle in the centerFrequencyBandwidth display. The polePlot and centerFrequencyBandwidth are both representations of the same instance of the twoPoleFilter class.

There are several number boxes in the samplePatch window which display in numeric form the same instance variables which are displayed in analog form by the samplePatch icon in the instrument window. Double-clicking on the Sound icon will message the IMW Signal Editor [Eckel 1990] which can display time and frequency domain representations of the sampled sound and provides a rich set of graphic signal

editing tools.

The twoPoleFilter representations and the Sound Icon representation are bound to pointer instance variables of the samplePatch class. When the samplePatch prototype icon is copied in the instrument window with the itention of placing a new samplePatch instance in the keymap, the intention is not simply to create a new instance of samplePatch but to create new instances of some of the objects pointed to, e.g. a new twoPoleFilter. The sound pointer in the samplePatch is more complex. A new sound should not be created but should the new sound pointer refer to the old sound or should it be initialized as void? Copy properities associated with pointer instance variables in Animal, allow these kinds of behavior to specified. Null representations can be specified to allow for the case of void pointers.

Animal Application Design

Animal encourages a rather free approach to application design. One can start by drawing a picture, then declare this picture to be a representation of a new or existing class. One can start by defining a new class then attach some existing representations to it. These representations may be "stolen" from other classes. One can create instances of a class and afterwards modify the class structure. This causes an automatic update of all existing instances of the class. One can add or modify representations of classes which have existing instances. The philosophy is incremental, unconstrained application prototyping through progressive refinement. Almost all class definition is accomplished through design of the class representation. The philosophy is, as much as possible, to provide the illusion of "drawing" the application.

In an environment oriented toward rapid prototyping it is important to support a high degree of reusability. Animal support two levels of reusability: reusability of objects inside a project; and reusability by easy sharing objects between projects.

Reusability inside a project is provided by an environment that encourages specification by cloning in every phase of the prototyping cycle, from the interface to the multiprocessor methods, and by providing an inheritance mechanism in the application data model.

Sharing objects between projects is supported through centrally maintained libraries for the different entities in an Animal application.

The Animal architecture has three kinds of entities that can be considered as modules to be shared between projects: the Classes (including method sources), the representations and the instance graphs or subgraphs.

The objects which are stored in a library are dearchived from the library when an application is dearchived. All applications using a library entity will always dearchive the most current version.

As previously mentioned all user code is written for the multiprocessor, none for the host Animal process. Main features of the environment are incremental method loading and an automated make utility.

Programming an Animal application involves writing methods for classes, where the classes have been defined graphical through manipulation of their representations. A method can be declared by placing a method representation--a kind of text box--inside a representation of the class.

The source code for methods can be edited by double-clicking on the representation of the method either in a class representation or in the class inspector. This opens an editor on the source file. If a user defined method has just been declared then a "do nothing" template method will be automatically generated with appropriate #includes, etc.

Animal automatically generates a Makefile for the application. This makefile has targets for specifying the compilation and loading of a single file or of all the files based on the standard Makefile dependencies.

Depending on the modifications which have taken place since the last compilation, Animal will either call the make utility with "all" as target, using the make policy to keep dependencies between files, or will use make to compile just the files that really need to be modified. This is to avoid unnecessary recompilation of all methods when a downward compatible class modification has been made--such as adding an instance variable.

Incremental loading is always mediated by the Animal process and requested by the make command through the use of a shell command. The makefile keeps track of which object files have been loaded and never reload an unmodified object file.

IMW Signal Editor

The Signal Editor supports viewing and editing of audio signals in time and frequency domains. The frequency domain tool, called SpecDraw, shows sonographs--"two-and-a-half" dimensional representations of time vs. frequency with amplitude mapped to grey levels--on which the user can design arbitrary polygons. The user can filter out the contents of the polygon or everything outside the polygon. In the future, arbitrary cut and paste of polygon regions will be supported.

IMW Recorder

The "Universal Recorder" application is intended to provide a multi-track recorder paradigm for spooling real-time event streams and signal streams to disk or memory. The recorder uses message and signal "probes" which can be attached to FTS objects to tap onto their message streams and, on playback, to regenerate these streams in a time-accurate manner.

Conclusion

The IMW is a large project which is still under development at the time of this writing. A contract has been signed with a U.S. manufacturer for the commercialization of the hardware and the base-level software (CPOS, math library, and basic development environement). Other software packages will be made available separately. It is hoped that these tools will provide an excellent environment for computer music performance and research.

[Intel 1989] i860 64-Bit Microprocessor Programmer's Reference Manual

[Lindemann, E., De Cecco, M. 1991] Animal: Graphical Data Definition and Manipulation in Real-Time. See this issue of CMJ.

[Puckette, M. 1991] Combining Event and Signal Processing in the MAX Graphical Programming Environment. See this issue of CMJ.

[Eckel, G. 1990] A Signal Editor for the IRCAM Musical Workstation. Proceedings of the International Computer Music Conference 1990, Glasgow.

[Smith, B. 1990] A Universal Recorder for the IRCAM Musical Workstation. Proceedings of the International Computer Music Conference 1990, Glasgow.

[Di Guigno, G., Gerzso, A. 1986] La Station de Travail Musical 4X - IRCAM

[Lindemann, E. 1987] DSP Architectures for the Digital Audio Workstation. AES Preprint - 83rd Audio Engineerig Society Convention, New York.

[Moorer, J.A. 1982] The Audio Signal Processor. CMJ, Vol. 6, No. 3.

[Puckette, M. 1991] FTS: A Real-Time Monitor for Multi-processor Music Synthesis. See this issue of CMJ.

[Lindemann, E., Dechelle, F., Smith, B., Starkier, M. 1991] The Architecture of the IRCAM Music Workstation. See this issue of CMJ.

[Viara, E. 1991] CPOS: A Real-Time Operating System for the IRCAM Music Workstation. See this issue of CMJ.

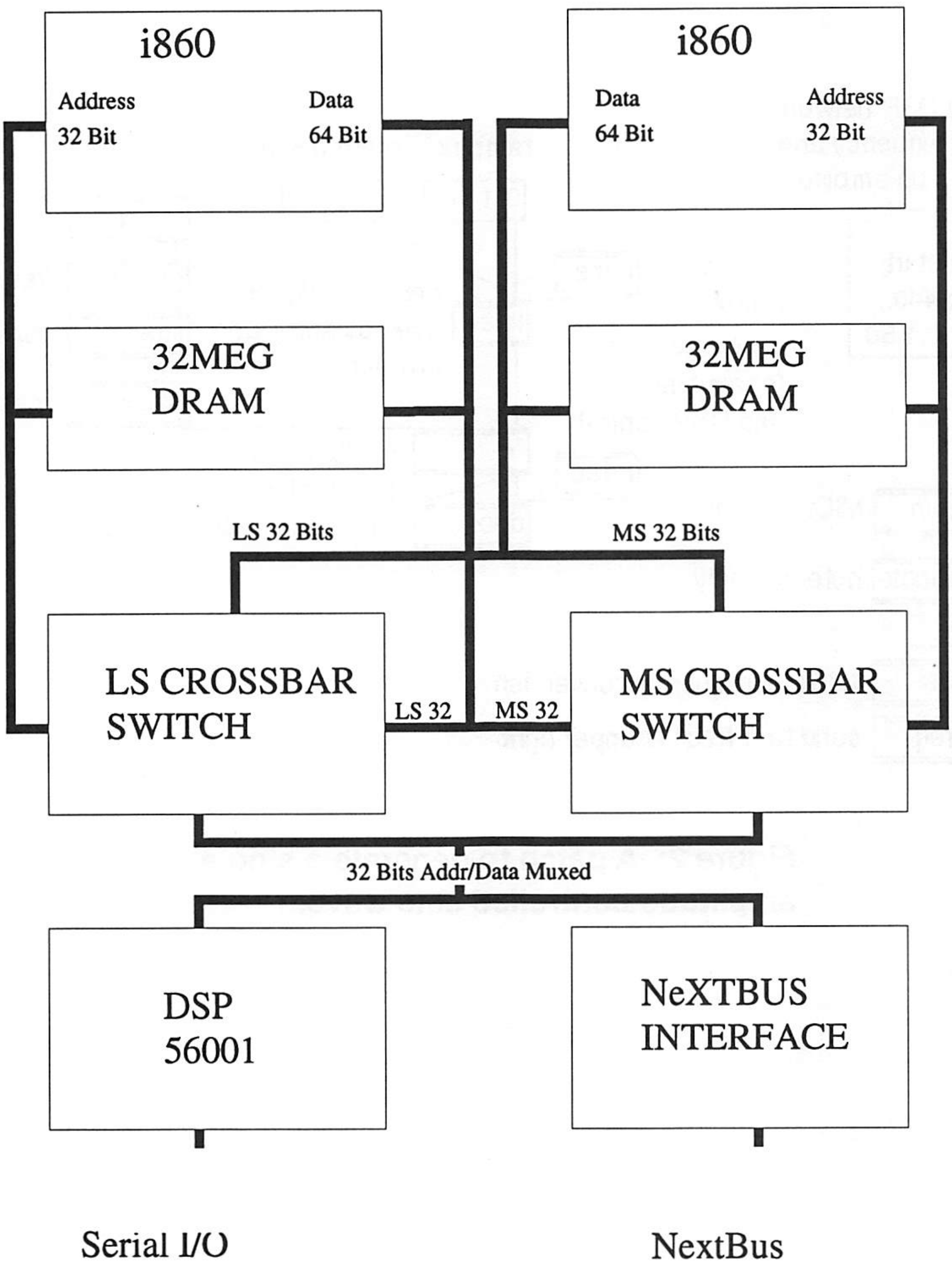


Figure 1: IMW Calculation Engine

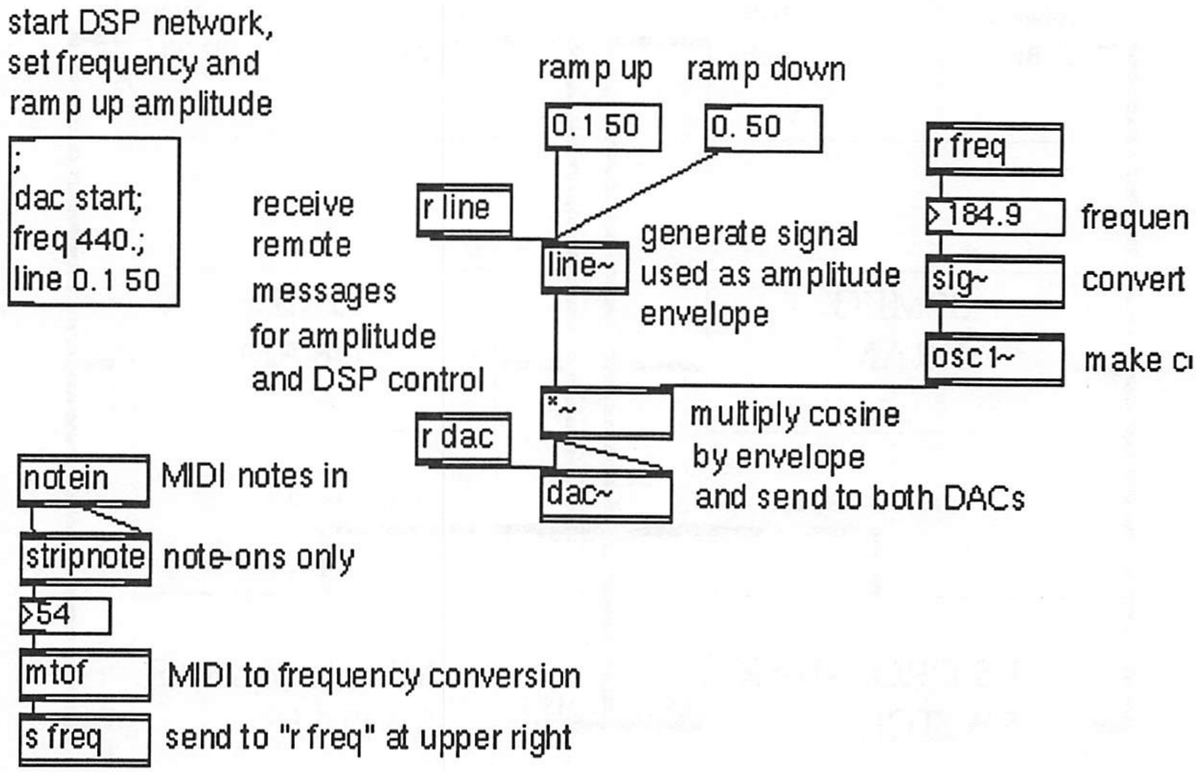


Figure 2: A patch to generate a single amplitude-controlled sine wave.

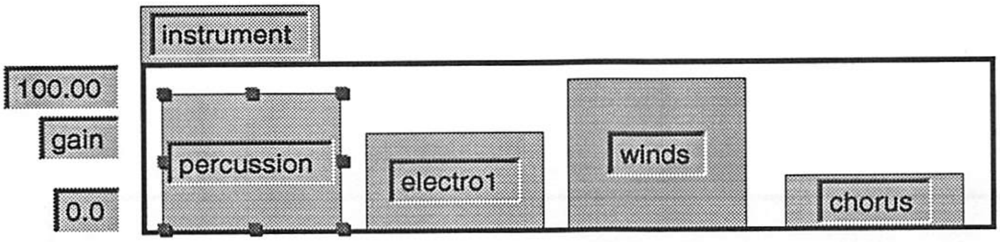


Figure 3: Orchestra Window

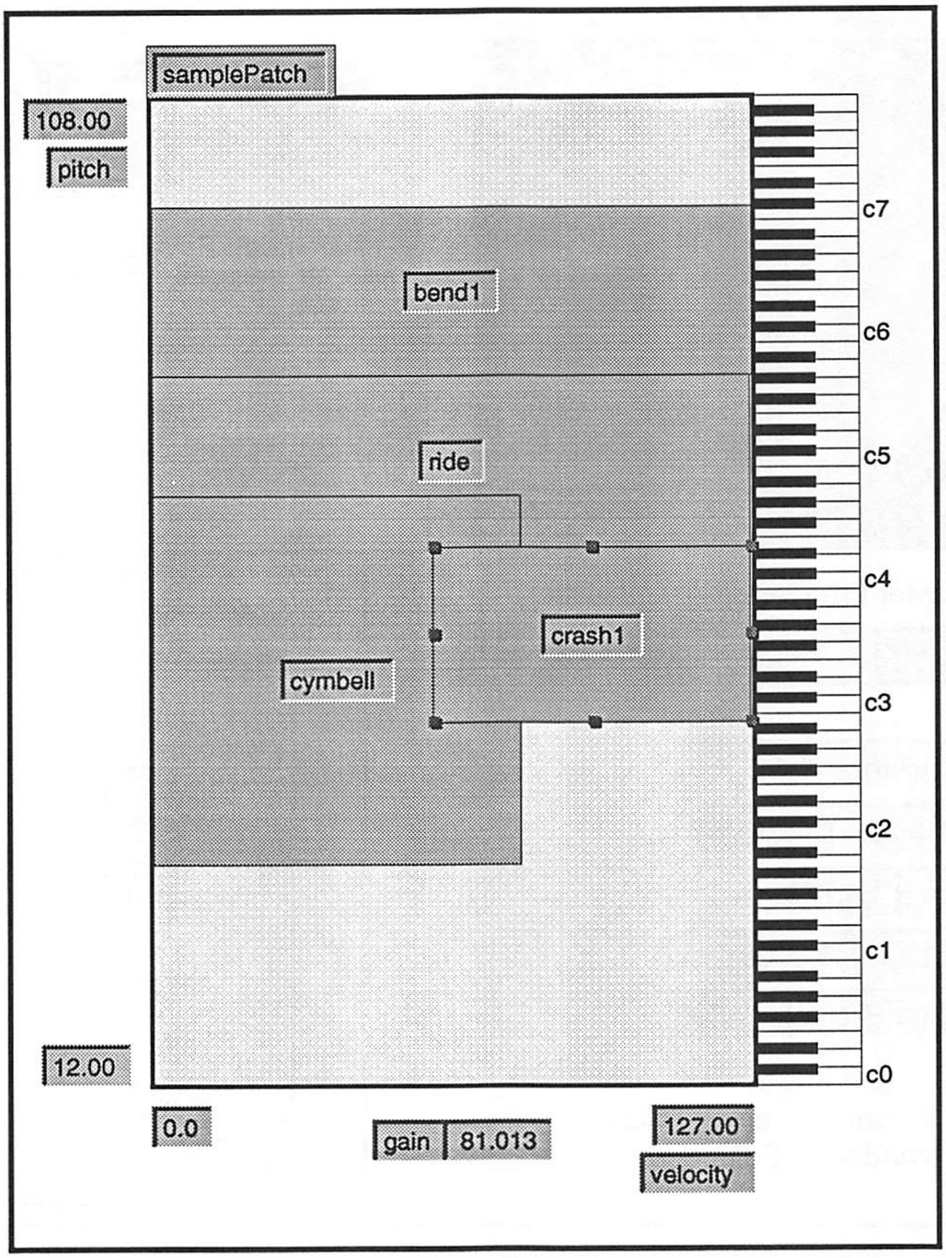


Figure 4: Instrument Window

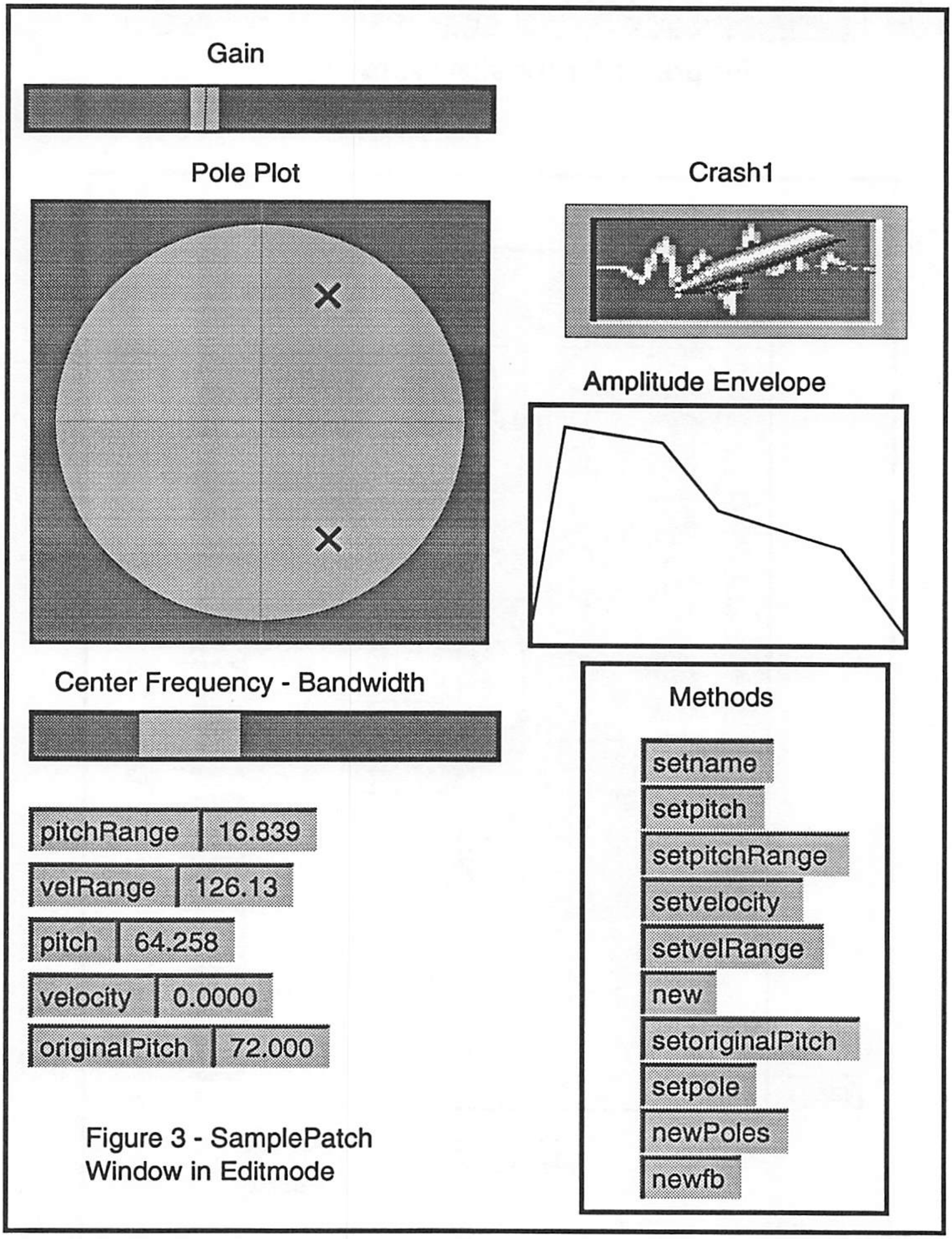


Figure 5: SamplePatch Window

Fast String Searching

Andrew Hume

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

`andrew@research.att.com`

Daniel Sunday

Johns Hopkins University/Applied Physics Laboratory
Laurel, Maryland 20723

`dan@aplexus.jhuapl.edu`

ABSTRACT

Despite substantial theoretical progress, there has been little improvement in practical string searching since the Boyer-Moore algorithm in 1977. And even Boyer-Moore took several years to infiltrate the UNIX® community. We describe a new framework for understanding a class of algorithms based on the skip loop variant of Boyer-Moore and describe various components that can be used in this framework. This leads to new algorithms that are substantially faster than the algorithms in current use. Furthermore, new algorithms can be synthesised automatically from the names of component pieces. The code for the algorithms is publicly available.

Introduction

Searching for a string in a body of text is one of the choice problems of computer science; it has spurred much theoretical work and is important in many applications. Surprisingly little of the theory has filtered down to the UNIX community, and slowly at that. Hume and Sunday[13] describe a framework for building fast algorithms. This paper briefly recounts that framework and some of its components, and describes a few of the resulting algorithms, and how to get the code.

Research in string searching has centered on reducing the number of character comparisons for the worst case. The standard problem is searching for all occurrences of a string of m characters in a string of n characters. A linear upper bound linear was discovered in 1977 for two algorithms. The Knuth-Morris-Pratt (KMP) algorithm[15] has a worst case bound of $2n-m$, tight for $m \geq 2$. The Boyer-Moore algorithm[4] first had a bound of $6n$[15] and this has been improved to a tight bound of $3n$[5], although a related algorithm by Apostolico and Giancarlo[2] has a bound of $2n-m$. Recently, Colussi, Galil and Giancarlo[6] derived an algorithm based on KMP and Boyer-Moore that improved the limit to $(4n-m)/3$.

More practical investigations of string searching, such as [11] and [17], have addressed different concerns, particularly the use of the runtime metric, and evaluating the typical, rather than the worst case, performance. These studies have almost always preferred the Boyer-Moore algorithm, although strangely enough, the slow form of the Boyer-Moore algorithm! The slow form, which we will call *classic BM*, was used for presentation and analysis of the algorithm. Boyer and Moore described a second form, which we call *fast BM*, as the preferred form for implementation. Despite this, and despite being much better in both run time and the number of character comparisons, fast BM has been largely neglected in the literature (notable exceptions are [19, 12, 11]).

From the public record, work on fast string searching in UNIX has been mainly as enhancements of or analogues to the *grep* programs. Early work utilising special purpose hardware support, for example, the `matchc` instruction on the VAX computers, was superceded by various implementations of Boyer-Moore. Bain released *bm* and *match* in 1985, as did Robbins and Mongiovi with *bgrep*. All these programs used straightforward implementations of classic BM and looked for one or more strings in a set of files. In 1986,

Woods[20] released a "peppy" version of *egrep* that used fast BM as a filter to eliminate lines that couldn't match. That BM code was packaged as a set of subroutines and released later in 1986. In 1986–7, Hume[12] restructured Research UNIX's *egrep* for improved I/O management and tighter integration of fast BM as a filter. Haertel improved Woods' fast BM in GNU *grep/egrep* in 1988[9]. The only work we know of unrelated to *grep* was in 1985; King and Macrakis[16] incorporated fast BM into GNU Emacs.

For the rest of this paper, we define the string searching problem as looking for all occurrences of a string *pat* of length *patlen* in another string *text* of length *textlen*. Code fragments are shown in C[14]. The main metric used in this paper is run time, or rather, the speed (in megabytes of text searched per second) for processing a standard test set. We will also report the number of character accesses, which is the sum of the number of comparisons and the number of uses in a skip loop. We will ignore preprocessing costs; this is typically negligible but it may dominate if *textlen* is small or if one is only looking for the first match and it is expected to be close to the start of *text*.

Testing methodology

Before describing the taxonomy and its components, we'll describe how we measured and compared components' performance.

The various algorithms were implemented in C using normal efficient programming techniques, for example, using register variables and character pointers instead of array indices. The test harness read the text to be searched and all the search words into memory before timing started. The text was then searched for each word sequentially. To gauge the dependence of the algorithms on the system type, the tests were run on a variety of systems listed below. The code was compiled without change using the compiler options shown below. Note that the systems were chosen as a diverse set of conveniently accessible machines representing most modern architectures, and not as representative of all existing systems.

system	compiler	description
`386`	`cc -O`	AT&T (Intel 80386)
`68k`	`lcc`	AT&T Gnot (Motorola 68020)
`cray`	`cc -O`	Cray XMP/28
`mips`	`cc -O3`	SGI 4D/380 (MIPS R3000)
`sparc`	`cc -O4`	Sun 4/260 (SPARC)
`vax`	`cc -O`	VAX 8550

The test results show the speed in MB of text searched per CPU second for each system, the average stride (step) of the skip loop through the input text and the percentage of input text characters accessed. The latter includes the accesses to step past mismatches (jump) as well as character comparisons between the pattern and the input text (cmp). All the tests used the same input text — a randomly selected 1MB subset of words from the King James Bible. Except for `cray`, the timings were all done on single-user systems with no other processes running. The mean of three runs was used; the average spread of the times was about 0.5%.

The standard test was looking for all occurrences of 500 unique words, 200 selected randomly from the unique words in the whole bible and 300 selected from the 1MB test subset. Of the 500 words, 428 were found in the test text, with 15228 matches in all. Word lengths varied from 2 to 16, the mean length was 6.95, and the standard deviation was 2.17. The other timing tests, for words of the same fixed length, used groups of 200 words or so selected randomly in the same fashion. We used the bible as the text to be searched because it is more representative of natural English text than the other convenient word lists (like dictionaries or online manual pages) and could be publicly released (at least, after permuting the order of words).

A taxonomy

The algorithms described here have a common structure:

```
for i over text do
begin
    skip loop
    match(text[i] with pat)
    i := i + shift(i,p)
end
```

Starting at a given position in *text*, there is first a *skip loop* (Boyer and Moore referred to this as the ''fast loop'') which quickly skips past obvious mismatches . When a possible match position is found, a *match algorithm* compares *pat* against *text* at that position, performing some action when a match is found. Finally, a *shift function* using i and p (where $text[i+p] \neq pat[p]$ after the match test) increments i and the algorithm progresses until it reaches the end of *text*. A full description of all components (some of which are omitted here) is given in [13]; the components used in this paper are described below.

Skip Loop Components	
none	the *skip loop* is omitted from the algorithm
sfc	search for first (leftmost) character in *pat*
slfc	search for least frequent character in *pat*
fast	search for last (rightmost) character in *pat* (fast BM)
ufast	portable, unrolled variant of *fast*
lc	least cost frequency dependent variant of *ufast*

The match component compares *pat* with *text* in some specific order. The order of comparison is significant; and some scan orders may have hardware or system support. Additionally, there may be a special *guard* test for a match with a specific (low frequency) character before starting the full match algorithm.

Match Algorithm Components	
fwd	forward linear scan (left to right)
rev	reverse linear scan (right to left)
om	optimal mismatch ascending frequency order
fwd+g	test *guard* before *fwd* scan
rev+g	test *guard* before *rev* scan

The final component, the shift function, cannot be arbitrarily mixed with other components. For example, using *d2* below depends on knowing the rightmost mismatch; thus the *rev* match must be used.

Shift Function Components	
inc	1
d1	BM's $delta_1(text[i+p])$
d2	BM's $delta_2(p)$
d12	BM's $max(delta_1(text[i+p]),\ delta_2(p))$
sd1	Sunday's $DELTA_1(text[i+patlen])$
md2	mini version of Sunday's $DELTA_2$ on the skip character
gd2	Giancarlo's generalised $delta_2(p)$
multiple	use maximum of several other shifts

We denote the maximum of multiple shifts by joining them with the ^ operator; for example, *d12* can be written *d1*^*d2*. Here are some classifications of algorithms drawn from the literature:

Algorithm	Ref	skip loop	match	shift
BM.ORIG	[4]	*none*	*rev*	*d1*^*d2*
BM.FAST	[4]	*fast*	*rev*	*d1*^*d2*
SFC	[11]	*sfc*	*fwd*	*inc*
SLFC	[11]	*slfc*	*fwd*	*inc*
QS	[18]	*none*	*fwd*	*sd1*

The notation for describing a particular algorithm, say BM.ORIG, is (skip=*none*, match=*rev*, shift=*d12*=*d1*^*d2*), or, more tersely, {*none*|*rev*|*d1*^*d2*}.

Component descriptions

The following component descriptions have been significantly condensed from [13].

Skip loops

• `none`

This does no skipping past obvious mismatches. Algorithms without a skip loop will always be relatively slow.

• `sfc`

Sfc (Search First Character) is a simple loop that looks for $ch = pat[0]$. Adding a sentinel to Horspool's code[11], that is, `text[textlen] = ch`, in order to terminate the loop is strongly advised.

As this skip loop is often supported in hardware, we also measured *sfcm*, a version that uses the library routine *memchr* to find *ch*. Although *memchr* is presumably as efficient as possible, all of the compilers we measured make a subroutine call to *memchr* rather than inserting the code inline.

• `slfc`

Horspool also described *slfc* (Search Least Frequent Character) which sets *ch* to the least frequent character, rather than the first character, in *pat* and thus spends more time in the skip loop. The *slfcm* variant uses *memchr* to find *ch*.

• `fast`

This is a straightforward implementation of fast Boyer-Moore. It combines two tests in the skip loop, one for end of text and one for a possible match, by using $delta_0$, identical to Boyer-Moore's $delta_1$ except that $delta_0[pat[patlen-1]]$ contains a sentinel value `LARGE` that causes the skip loop to halt when a possible match position occurs. Our implementations is based on that by Woods[20]. Written with character pointers rather than arrays, and so that *s* is aligned with the right hand end of *pat*, the skip loop is

```
while((s += d0[*s]) < e)
        ;
```

Most C compilers, with optimising enabled, generate very good code for this loop. For example, compilers for VAX computers typically generate a four instruction loop.

• `ufast`

The `LARGE` sentinel in *fast* causes problems on segmented architectures and with arithmetic overflow on pointer addition. In addition, the ANSI C standard[1] specifies pointers may be compared only if they point inside or just after the same array of characters. These problems are both avoided by an alternate scheme devised by Haertel for the Free Software Foundation's *egrep*[9], although his primary motivation was to support loop unrolling[10]. It uses a sentinel of zero (and thus is identical with BM's $delta_1$) so that the skip loop becomes:

```
k = 0;
while((k = d0[*(s += k)]) && (s < e))
        ;
```

Because of the double test, this runs a little slower than the *fast* loop but this is easy to correct by conventional tuning tricks; in particular, by unrolling the loop. Experimentation showed that 3-fold unrolling was the best compromise across systems.

• `lc`

The speed of *fast* and *ufast* depend on how much time is spent in the skip loop. The cost of breaking out of the loop is such that it is worthwhile investigating how to maximise the amount of time in the skip loop. Consider searching for the word "baptize" in English text. Repeatedly, the "e" will match and terminate the skip loop. However, if the skip loop was looking for the "z", the skips would be a little smaller but the "z" would almost never match and the loop would run faster overall. *Lc* uses the skip character that minimises the expected runtime of the search.

While *lc* only averages about 3-5% faster than *ufast*, there are occasional speed ups of up to 15%. For the case of alphabets with a small variance in the character frequencies, we would expect *lc* to be less effective. Otherwise, for the case of very long patterns, we expect *lc* to be more effective.

• Summary

The performance figures for the various skip loop components were measured with match=*fwd* and shift=*inc*.

Algorithm	Execution Speed (MB/s)						Statistics	
	`386`	`sparc`	`mips`	`vax`	`68k`	`cray`	step	cmp+jump
none	0.15	1.09	2.50	0.57	0.47	0.78	1.00	1041392 (104.1%)
sfc	0.58	2.99	6.09	1.83	1.36	2.28	1.00	1041397 (104.1%)
sfcm	0.97	1.77	4.24	3.13	0.92	2.84	1.00	1041397 (104.1%)
slfc	0.62	3.18	6.27	1.92	1.43	2.39	1.00	1023043 (102.3%)
slfcm	1.29	1.90	4.54	4.08	1.07	5.23	1.00	1023044 (102.3%)
fast	2.42	6.73	10.92	5.13	3.41	7.68	5.22	202619 (20.3%)
ufast	2.66	7.11	12.52	5.81	4.21	9.21	4.95	213048 (21.3%)
lc	2.27	7.13	12.57	5.71	4.28	9.23	4.93	212909 (21.3%)

None is the obvious naive algorithm. Ignoring the preprocessing step, *slfc* is usually better than *sfc*. The benefits of using the *memchr* library routine are system specific.

The *fast*, *ufast*, and *lc* skip loops are clear winners. The *ufast* loop seems the best balance between ease of coding, portability, and performance.

Match Algorithms

In this section, we describe different techniques for determining if *pat* fully matches *text* at its current position. After all but the *none* skip loop we know that a particular text character already matches; and most match algorithms can use this information. The choice of match algorithm is often dependent on the shift function one wants to use. Some shifts, for example *d1* and *d2*, require knowing the rightmost mismatch point. In this case, one could use *rev* but not *fwd*.

• `fwd`

The simplest matching algorithm simply compares *pat* with *text*[*i*] from left to right, thus yielding the leftmost mismatch. There is often special system and/or hardware support for *fwd*.

• `rev`

The reverse scan is the obvious counterpart to *fwd*, comparing from right to left and thus, yielding the rightmost mismatch. It is rare for a system to have special support for reverse character comparison.

• `om`

According to Sunday[18], we will make the optimal (minimal) number of match comparisons if they are done in ascending frequency order. However, unless you generate code on the fly, the implementation inherently involves double indirection and runs slower on most architectures.

• `guard`

For our test set, the average number of characters compared in the *om* loop is 1.05. Thus, testing for the rarest character of the pattern before doing a full match gains us 95% of the benefit of the *om* match. We set *rarest* such that *pat*[*rarest*] is the *pat*[*k*] (other than the skip loop character) with lowest frequency value. The guard test is

```
if(s[rarest] != pat[rarest])
        goto mismatch;
```

The guard can be combined with any other match algorithm. A guard is denoted by "+*g*", so *fwd*+*g* is a guard test before a *fwd* match scan. Match algorithm properties such as yielding the rightmost mismatch

may be abrogated by using a guard.

• Summary

The match algorithms were measured with skip=*ufast* and shift=*inc*. We also measured *fwdm*, a version of *fwd* that used the library *memcmp* routine.

Algorithm	Execution Speed (MB/s)						Statistics	
	`386`	`sparc`	`mips`	`vax`	`68k`	`cray`	step	cmp+jump
fwd	2.66	7.11	12.52	5.81	4.21	9.21	4.95	213048 (21.3%)
rev	2.66	6.97	12.60	5.80	4.14	9.09	4.95	215464 (21.5%)
om	2.32	7.06	12.37	5.78	4.15	9.48	4.95	212791 (21.3%)
fwdm	2.59	6.56	12.14	4.78	3.23	8.31	4.95	213048 (21.3%)
fwd+g	3.04	7.30	12.64	6.10	4.35	9.67	4.95	203226 (20.3%)
rev+g	3.04	7.30	12.65	6.12	4.34	9.66	4.95	203425 (20.3%)
fwdm+g	3.04	7.25	12.85	5.87	4.12	9.57	4.95	203226 (20.3%)

The *fwdm* match (using *memcmp*) is uniformly inferior, which is not surprising as the average number of characters compared is only 1.05. *Fwd+g* seems a clear winner whenever it can be used.

Shift functions

In this section, we describe the different shift functions. Most require knowing where the mismatch occurred and some require knowing the rightmost mismatch.

• `inc`

This simply increments the current text pointer by one. It is simple and fast to compute but without a good skip loop, *inc* yields a slow algorithm.

• `d1`

This is the Boyer-Moore $delta_1$ and requires knowing the rightmost mismatch. The value of $delta_1[c]$ is the distance of *c* from the end of the pattern, or *patlen* if *c* is not in *pat*. This increment applies at the point of mismatch and thus may sometimes yield a nonpositive shift. In this latter case, use an overall shift of 1.

• `d2`

This is the Boyer-Moore $delta_2$ and requires knowing the rightmost mismatch. It is used with *d1* to give a linear bound on the worst case search behavior. The implementation is recondite; we used the one from Smit[17].

• `d12`

This is the shift function used by Boyer and Moore. It is simply the maximum of the *d1* and *d2* shifts.

• `sd1`

Sunday[18] observed that if the right hand end of *pat* is aligned with $text[r]$, then after the shift, some character in *pat* must align with $text[r+1]$, and so we can define a shift function $DELTA_1$ based on $text[r+1]$. $DELTA_1$ is actually $(delta_1 + 1)$ and always yields a positive shift amount. This shift is noteworthy because it is independent of where any mismatch occurs; thus, arbitrary and nonsequential scan orders can be used for the match algorithm.

• `md2`

If we use a skip loop other than *none*, we know a character in *text* has matched the skip loop character *c* and we can certainly shift *pat* by the amount needed to bring the next leftmost occurrence of *c* to the same position in *text*. This shift dramatically outperforms *inc*; it is just as efficient and the amount is typically a little less than *patlen*. For our test set, the average *patlen* is 6.95 and the average *md*2 shift is 6.23.

• `gd2`

The *gd2* shift is a generalisation of the $delta_2$ shift by Giancarlo[8] by incorporating the $delta_1$ character heuristic. The preprocessing, $O(m \times | alphabet |)$ in time and space, is substantially greater than the other shift functions. In return, the *gd2* shift function performs well on our test set but is astonishingly fast for long patterns.

• `multiple`

Combining two or more shifts seems a good idea but in practice, the additional overhead outweighs the advantages of slightly larger shifts.

• Summary

The following timings had skip=*ufast* and match=*rev*.

Algorithm	Execution Speed (MB/s)						Statistics	
	`386`	`sparc`	`mips`	`vax`	`68k`	`cray`	step	cmp+jump
inc	2.66	6.97	12.60	5.80	4.14	9.09	4.95	215464 (21.5%)
d1	2.01	6.82	12.46	5.56	4.02	8.84	5.12	218923 (21.9%)
d2	2.05	6.84	12.36	5.57	3.99	8.82	5.00	223531 (22.4%)
d1^d2	2.44	6.80	12.32	5.49	3.88	8.88	5.14	218203 (21.8%)
sd1	2.68	7.02	12.72	5.87	4.18	9.11	5.22	204908 (20.5%)
md2	2.72	7.19	12.96	5.97	4.31	9.46	5.18	216368 (21.6%)
gd2	2.44	6.93	12.47	5.44	3.89	8.85	5.21	205000 (20.5%)

For the typical case of searching for shortish words in natural language text, *md2* is the fastest across all the systems we measured. It is fast, compact and easy to precompute.

Recommended searching algorithms

For general purpose use, we recommend the algorithms TBM and LC. TBM (Tuned Boyer-Moore), or $\{ufast | fwd + g | md2\}$, is compact and fast and can be made independent of frequency data by eliminating the guard. LC (Least Cost), or $\{lc | fwd + g | md2\}$, differs from TBM only in the skip loop used; the *lc* loop makes it a bit faster in some instances. Without frequency data, TBM and LC are equivalent to $\{ufast | fwd | md2\}$.

Although we consider TBM and LC the best general purpose algorithms, they may not be optimal for a particular architecture. We show below the fastest programs we found (by educated guesses, not exhaustive search) for each of the systems we tested. The ''+c'' suffix denotes that the type of the skip vector (such as `d0`) was an 8 bit, rather than a 32 bit, integer.

Algorithm	Execution Speed (MB/s)						Statistics	
	`386`	`sparc`	`mips`	`vax`	`68k`	`cray`	step	cmp+jump
TBM	**3.11**	7.51	12.98	6.25	4.53	**10.08**	5.18	204199 (20.4%)
{lc+c|fwd+g|md2}	2.33	**8.15**	**14.20**	5.47	4.17	8.07	5.16	204592 (20.5%)
LC	2.45	7.54	13.10	**6.31**	**4.57**	10.06	5.16	204592 (20.5%)

We are interested in any new components and in the best programs for various systems; please send details to the authors, preferably by electronic mail.

Discussion

From a practical point of view, the primary performance characteristic of a string search algorithm is speed in terms of CPU time. The primary metric used by theorists is character comparisons, but it is easy to design algorithms which do fewer character comparisons and yet have a slower runtime. For example, on `386` and `vax`, $\{sfcm | fwd | inc\}$ is much faster than $\{none | rev | d12\}$ (classic Boyer-Moore) despite doing three times as many character comparisons.

We will compare the performance of TBM and LC against some popular algorithms: BM.ORIG $\{none | rev | d12\}$, QS $\{none | fwd | sd1\}$[18], and BM.FAST $\{fast | rev | d12\}$.

Algorithm	Execution Speed (MB/s)						Statistics	
	`386`	`sparc`	`mips`	`vax`	`68k`	`cray`	step	cmp+jump
BM.ORIG	0.36	2.38	5.04	1.10	0.95	2.05	5.42	382579 (38.3%)
QS	0.79	4.16	8.51	2.62	2.03	3.59	6.24	172365 (17.2%)
BM.FAST	2.18	6.47	10.75	4.60	3.22	7.52	5.42	208169 (20.8%)
TBM	3.11	7.51	12.98	6.25	4.53	10.08	5.18	204199 (20.4%)
LC	2.45	7.54	13.10	6.31	4.57	10.06	5.16	204592 (20.5%)

Does this comparison depend on the pattern lengths? Figure 1 shows the effect of the pattern length relative to BM.FAST. Each test looked for 200 randomly selected words of the specified length. The test was run on just one system (`mips`). Note that QS generally runs slower than the other algorithms despite always making fewer character comparisons.

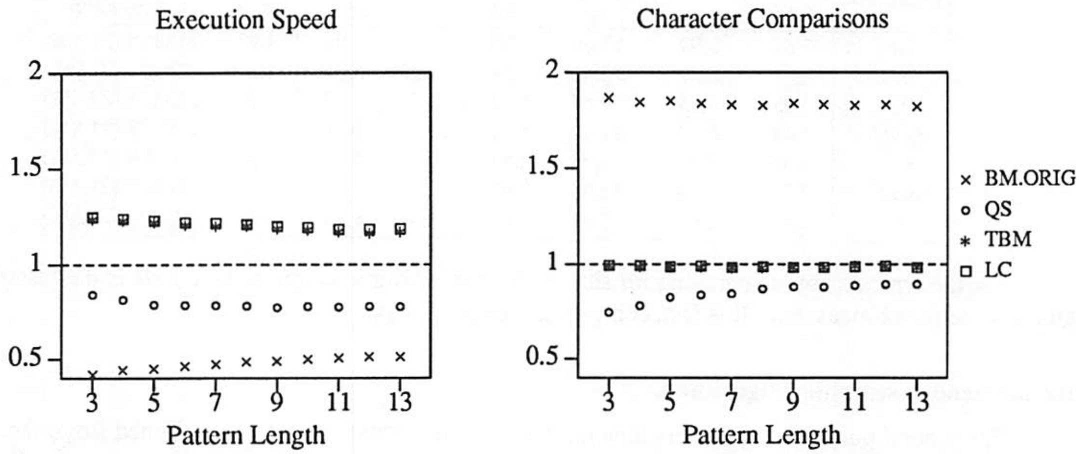


Figure 1. Performance of the main algorithms relative to BM.FAST

Some of the algorithms described above depend on frequency distributions. How sensitive is their performance to the accuracy of the distribution used? The answer is surprising; the performance is almost independent of the distribution. The performance of TBM and LC was measured for four different frequency distributions: (a) optimal (the measured distribution of *text*), (b) pessimal (1–(a)), (c) distribution for 7.7MB of formatted manual entries, and (d) distribution from 11.2MB of C program text. The maximum difference in speed between the four distributions was only 0.75%! This is partially explained by our standard test that averages out the occasional real winners (and less frequent losers). In practice, any plausible distribution based on English will do just fine.

Finally, we measured various shift functions for the case of long pattern strings. Normally, these are rare but in some domains, such as searching genetic databases, they are commonplace. The patterns were DNA sequences from 10 up to 250 characters long and the data searched was a portion of the GENBANK DNA database[3]. The results are shown in Figure 2. The prodigious speed of the *gd2* shift clearly deserves further attention.

Automatic generation of algorithms

The taxonomy presented here suggests the possibility of automatically synthesising algorithms from skip, match and shift components. The program *gen* does this; it takes arguments denoting the skip, match and shift components and produces about 80–140 lines of C for the preprocessing and execution routines on its standard output.

The process of synthesising the C is straightforward. There is a database mapping component names to the files containing the corresponding code fragments. The fragments are then spliced together according to a table compiled into *gen*. We discuss below the database and the structure of the code fragments.

The component database

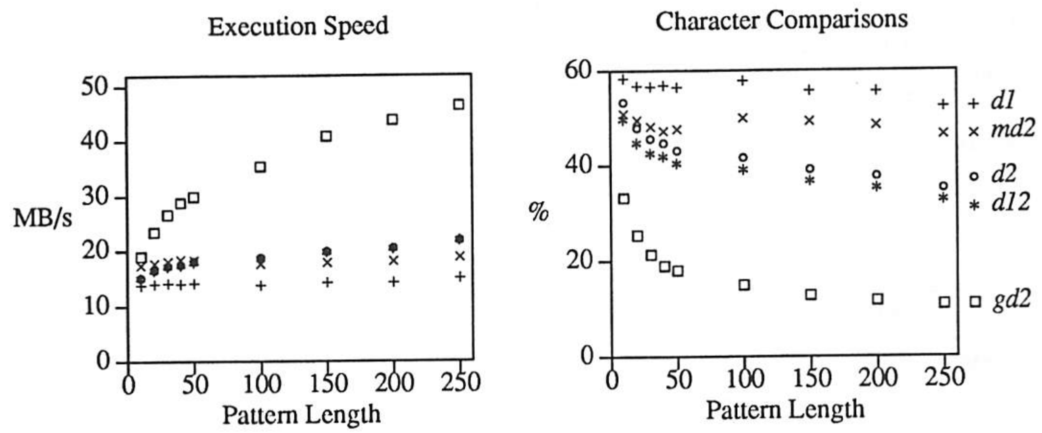


Figure 2. Performance of *shift* functions in searching DNA strings

This grandiose sounding object is a simple text file; here are some sample lines:

```
# main body template
body    proto/body
# skip components
uf      proto/uf3              nm
sfc     proto/sfc              0m
no      proto/no               n
# match components
fwd     proto/fwd              b
rev     proto/rev              rb
revr    proto/revr             r
revl    proto/revl             r
# shift components
inc     proto/inc
md2     proto/md2              m
```

The format is simple — the component name, the filename for the C fragments and optional attributes. One attribute relates to database management. If either the match or shift component have the `b` attribute, for example, `rev`, it means there are two versions of the component, with the suffices `l` and `r` respectively, which can be used if the skip component has either the `0` or `n` attribute (explained below). (For no compelling reason, our `no` skip loop anchors the right hand end of the pattern rather than the more obvious loop that anchors the left end.)

Each file referred to above consists of an initial optional comment followed by a sequence of fragments, each starting with *$fragment_name*. As an example, the file for the Boyer-Moore $delta_1$ shift *dl* is shown in Figure 3. The various fragments are combined together according to a template compiled into *gen*. For example, the code that performs the shift is in the `exec` fragment, the variables used are declared in the `exec decls` fragment, the corresponding fragments used to set up the skip table are in `init` and `init decls` respectively, and finally, the skip table itself is declared in `decls` (this is inserted into the *pat* structure described below).

Model of execution

The output of *gen* is C source for a preprocessing routine *prep* and an execution routine *exec*. A structure named *pat*

```
static struct {
        CHARTYPE pat[MAXPAT];
        int patlen;
} pat;
```

is the sole means of communication between *prep* and *exec*. For example, the string being searched for is stored in the buffer `pat.pat`. Because of the timing tests and simplicity, the code uses fixed sized buffers

```
        the Boyer-Moore delta1 shift function
$decls
        Tab delta1[256];
$init decls
        register Tab *d1;
        register j1;
$init
        d1 = pat.delta1;
        for(j1 = 0; j1 < 256; j1++)
                d1[j1] = m;
        for(j1 = 0; j1 < m; j1++)
                d1[base[j1]] = m-1-j1;
$exec decls
        register Tab *d1 = pat.delta1;
        register k1;
$exec
                k1 = q+d1[*q]-RH;
                if(k1 < 1)
                        k1 = 1;
#ifdef  STATS
                stats.step[k1]++; stats.jump++;
#endif
                s += k1;
```

Figure 3. Code file for *d1*

and simply increments a variable when a match is detected. These restrictions can be easily fixed for real applications and with care, the preprocessing code can be merged with the execution code.

Prep has this skeleton

```
prep(base, m)
        CHARTYPE *base;
{
        CHARTYPE *skipc;
}
```

where *base* points to the string to be searched for and *m* is the length of the string. The string is copied into *pat.pat* and *skipc* is set to point at the character in *pat.pat* that the skip component loops on or zero otherwise.

The execution routine has this skeleton

```
exec(base, n)
        CHARTYPE *base;
{
        CHARTYPE *s, *p, *q;
        int s_offset;
}
```

where *exec* is to search *n* bytes starting at *base*. The skip loop is to use *s* to step through the text. Just before the match component, *s* points to either the byte corresponding to `pat.pat[0]` (property `0`) or to `pat.pat[m-1]` (property `n`) or somewhere in between, in which case `s[s_offset]` corresponds to `pat.pat[0]`. If the skip component has attribute `m` then `*s` matches its corresponding character in *pat.pat* after the skip loop terminates.

The match component must not alter *s*. After the match code completes, *p* and *q* must point at the bytes in *pat.pat* and *base* respectively which mismatch. If there is no mismatch, *p* must be set to `&pat.pat[-1]`. If the match component has attribute `r`, it yields the rightmost mismatch (left of *s*).

Finally, the shift component must increment *s* by at least 1. If it has the `r` attribute, it requires that the match code yield the rightmost mismatch. If the shift component has the `m` attribute, then the skip component must have the `m` attribute.

Finally, any component may indicate that it needs one or more header files. The `f` attribute includes `"freq.h"` which should initialise the array

```
double freq[256];
```

such that $freq[c_0] < freq[c_1]$ iff c_0 is less frequent than c_1. The `s` attribute includes `"sd.h"` which defines an array of expected skip loop amounts; see [13] for details.

Summary

The scheme described here is quite specific to the domain of synthesising fast string search codes according to our taxonomy. *Gen* is small, less than 400 lines of C source, and took only three hours to write. Including `l` and `r` forms, there were 16 skip loops, 15 match algorithms, 9 shift functions; extracting these code fragments from existing source took another four to five hours. Of the 37 algorithms we synthesised and measured, only four were slightly hand tuned (via *ed* scripts) from the output from *gen*, and three of those were simply to reduce the number of register variables.

Getting the code

All the programs, word lists and most of the testing apparatus used in this paper are available electronically via *ftp* (login as user `netlib` on `research.att.com`) or *netlib*[7]. *Netlib* works by sending electronic mail messages to a server. For example, the message

```
send index
```

will cause generic information on other packages and servers to be sent to you by return mail.

The material in this paper is available from the `stringsearch` package. The components at the time of writing are

`index`	summary of what's available
`bmsrc`	test harness and all algorithms
`bmdata`	bible text and wordlist
`bmctl`	scripts to generate performance data
`bmbio`	DNA text and wordlists

Both `bmdata` and `bmbio` are large; `bmsrc`, `bmctl`, and `bmdata` are sufficent to replicate the main performance measurements. For example, with *netlib* you would type something like

```
mail netlib@research.att.com
send bmsrc bmdata bmctl from stringsearch
.
```

We have made the code and data sets available primarily for two reasons. Firstly, having the source for these algorithms conveniently available can only raise the standard of software that does string searching. Some of the components, such as $delta_2$, are subtle and hard to implement from scratch. Secondly, we would like to start a trend of publishing (electronically) sufficient information to allow someone other than the author(s) of a string search paper to reproduce the results, or compare a new algorithm with an existing one on the same test data.

Conclusion

There is a significant amount of code that does string searching over large amounts of text. Historically, most of this code has tried to use hardware assists or classic Boyer-Moore. We have described a framework for synthesising algorithms with much better performance. In particular, TBM and LC are portable, compact algorithms that are nearly twice as fast as classic BM.

In order to promote the use of these faster algorithms, we have made their C implementations publicly available. We also hope to foster experimentation and comparisons with other algorithms by also making our evaluation environment, both datafiles and testing programs, publicly available.

Lastly, we acknowledge that some users have special needs not met by general purpose, portable algorithms. To help these users, we offer an algorithm synthesiser and a variety of components; this may not solve the problem at hand but gives a framework for exploring possible solutions.

Acknowledgements

James Woods sparked Hume's interest in the Boyer-Moore algorithms in 1986 and his management, particularly Al Aho, has indulged his fixation since. Sunday would like to thank JHU/APL for providing him with the resources to pursue this research.

References

1. ANSI, *Programming Language – C (X3.159-1989)*, American National Standards Institute, New York (1989).

2. Apostolico, A. and Giancarlo, R., 'The Boyer-Moore-Galil String Searching Strategies Revisited,' *SIAM J. Comput.* **15**(1), pp. 98-105 (February, 1986).

3. Bilofsky, H. S. and Burks, C., 'The GenBank® Genetic Sequence Data Bank,' *Nucl. Acids Res.* **16**, pp. 1861-1864 (1988).

4. Boyer, R. S. and Moore, J. S., 'A Fast String Searching Algorithm,' *Comm. ACM* **20**(10), pp. 262-272 (October 1977).

5. Cole, R., 'Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm,' Technical Report 512, Computer Science Dept, New York University (June, 1990).

6. Colussi, L., Galil, Z., and Giancarlo, R., 'The Exact Complexity of String Matching,' *31st Symposium on Foundations of Computer Science* **I**, pp. 135-143, IEEE (October 22-24, 1990).

7. Dongarra, J. J. and Grosse, E., 'Distribution of Mathematical Software Via Electronic Mail,' *Comm. ACM* **30**(5), pp. 403-407 (May 1987).

8. Giancarlo, R. (December 1990). Personal communication.

9. Haertel, M., *GNU e?grep*, Usenet archive `comp.sources.unix`, Volume 17 (February, 1989).

10. Haertel, M. (December 1990). Personal communication.

11. Horspool, R. N., 'Practical Fast Searching in Strings,' *Software—Practice and Experience* **10**(3), pp. 501-506 (1980).

12. Hume, A., 'A Tale of Two Greps,' *Software—Practice and Experience* **18**(11), pp. 1063-1072 (November, 1988).

13. Hume, A. and Sunday, D., 'Fast String Searching,' *Software—Practice and Experience* **?**(?), pp. ?-? (1991).

14. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ (1988).

15. Knuth, D. E., Morris, J. H. Jr, and Pratt, V. R., 'Fast Pattern Matching in Strings,' *SIAM J. Comput.* **6**(2), pp. 323-350 (June 1977).

16. Macrakis, S. (February 1991). Personal communication.

17. Smit, G. D. V., 'A Comparison of Three String Matching Algorithms,' *Software—Practice and Experience* **12**(1), pp. 57-66 (1982).

18. Sunday, D. M., 'A Very Fast Substring Search Algorithm,' *Comm. ACM* **33**(8), pp. 132-142 (August 1990).

19. Woods, J. A., *More Pep for Boyer-Moore Grep*, Usenet netnews group `net.unix` (March 18, 1986). Also Usenet archive `comp.sources.unix`, Volume 9 (March, 1987).

20. Woods, J. A., *More Pep for Boyer-Moore Grep*, Usenet archive `comp.sources.unix`, Volume 9 (March, 1987).

Andrew Hume is in the Computing Science Research Center at AT&T Bell Laboratories. Like all *real* computer scientists, he has written CAD tools, his own make, made *grep*, *fgrep* and *wc* go really fast and built a file backup system (files check in but they never check out). He has a compulsion to make things go fast, match regular expressions, and build/draw polyhedra.

Dan Sunday is a mathematician at the Johns Hopkins University Applied Physics Laboratory (JHU/APL), and works as the designer and developer of software systems. He is currently the Lead Software Engineer on a project developing networks of color display systems for the US Navy.

SFIO: Safe/Fast String/File IO

David G. Korn, att!ulysses!dgk
K.-Phong Vo, att!ulysses!kpv

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes *Sfio*, a new input/output library, that can be used as a replacement for *Stdio*, the C language standard I/O library. *Sfio* is more complete, consistent, and efficient than *Stdio*. New facilities are provided for convenient, safe and efficient manipulation of data streams. An *Sfio* stream may be entirely memory resident or it may correspond to some actual file. Alternative I/O disciplines can be applied to a stream to customize its behavior with respect to data transformation and exception handling. Stream pools can be maintained to guarantee stream synchronization when switching streams for I/O operations. Separate streams can be stacked on one another to make new virtual streams. Both source and binary compatibility packages are provided allowing *Stdio*-based programs to benefit from the new library without recoding. *Sfio* has been used successfully in a number of applications including many rewritten standard system utilities. Benchmark timings show that *Sfio* performs very well against *Stdio* and standard utilities can gain substantial performance improvement when based completely on *Sfio*.

1. Introduction.

The C language does not have any input/output facilities. I/O operations are typically negotiated via system or function calls. Most C programs routinely use the standard I/O library, *Stdio* [KR]. The library was written by Dennis Ritchie in the mid 1970's to replace an earlier library called the portable I/O library. *Stdio* was a part of UNIX[1] Version 7 which is the ancestor of virtually all current UNIX systems. Central to *Stdio* is the idea of a buffered stream that models an input or output file. Application code can read or write arbitrary chunks of data with confidence that the underlying buffering scheme will perform efficiently. Beyond efficiency concerns, *Stdio* also provides other functions to read and write various types of data included formatted records.

Today the standard I/O library is ubiquitous in C programming environments. Features from diverse versions have been amalgamated in the ANSI C standard [AN]. Despite its success, there are recognized deficiencies in *Stdio* both in its design and various implementations. For example, the library does not provide an efficient way to transfer large amount of data between streams. This weakness causes many standard system utilities to revert to direct use of the underlying system calls. The poor performance of *Stdio* for line-oriented processing was addressed in the design of yet another I/O library, *Fio*, done by Andrew Hume for the 9th Edition UNIX system [Hu]. The interface of *Stdio* is incomplete, prompting a number of attempts over the years to extend it. The ability to perform both input and output on a stream opened for both read and write was added in the early 1980's by Andrew Koenig. A few years later, the ability to tune buffer size and set line buffering mode was added in the BSD UNIX version. In our own experiences, a frustrating problem with *Stdio* is the lack of any facility for graceful handling of exceptions. A bad case for most current versions of *Stdio* is when the `write()` system call is interrupted. Most of the time, this would cause a loss of data. Equally perplexing is the lack of any facility for synchronizing buffers of streams that share the same underlying file, e.g., `/dev/tty` and the streams `stdin`, `stdout` and `stderr`. Thus, there is no way to guarantee serialization of I/O

1. UNIX is a trademark of AT&T Unix System Laboratory

operations short of flushing the buffer after each operation.

The *Sfio* library was written partly to overcome *Stdio*'s deficiencies. One may ask why not just rewrite *Stdio* and maintain if not binary compatibility, at least source compatibility? One of us, David Korn, tried this tack a number of years back. Numerous link editing problems occured because of name clashing. A major problem is the use of a fixed array of streams in *Stdio*. This makes it impossible to extend the stream data structure without violating binary compatibility. This limits the usefulness of a new implementation unless one can recompile the entire system and its host of applications. More important, preserving the *Stdio* interface means also preserving its various problems. We feel that a better interface could be designed that would avoid inconsistency, reduce redundancy, and be easily extensible. The good news is that by providing a completely new interface that occupies a different name space from that of *Stdio*, we are able to build both source and binary compatibility packages to *Stdio*. On a system with shared libraries, the compatibility packages can be installed to provide transparent access to *Sfio*.

The remainder of the paper will go into some details on the design and implementation deficiencies in *Stdio*. Then, the motivations behind *Sfio* and its main features are discussed. Benchmark data comparing the performance of *Sfio*, *Stdio* and *Fio* is presented. This includes data comparing standard versions of system utilities and our reimplementations which exploit the new abstractions provided in *Sfio*. Finally, the manual pages for *Sfio* are attached for detailed descriptions of the available functions.

2. The In-Problems with Stdio

There are many problems with the current design and implementations of *Stdio*. For reasons that should become clear soon, we call them the *in-problems*. In the following discussion, the reader should keep in mind that there are many versions of *Stdio* and certain problems may apply to only particular versions.

- *Incorrectness:* A design defect in *Stdio* is that it does not check the type of a stream when I/O operations are performed. For example, it is perfectly ok to open a stream for read only, then write data into it. Here is an example:

  ```
  f = fopen("data_file","r");
  getc(f);
  putc('a',f);
  ```

  Of course, only the data in the internal buffer of the stream `f` is changed when `putc()` is issued, not the underlying file. However, interesting consequences can arise in the application code.

  At the implementation level, the library also suffers from a number of defects. The case that caught our attention is when a system call gets interrupted by a signal. Here is an example:

  ```
  alarm(1);
  fwrite(buf,1,1000000,stdout);
  ```

  In this code fragment, the buffer being written out greatly exceeds the internal buffer of the standard output stream. Therefore, from time to time, a `write()` system call will be issued to flush data from the internal buffer. If the alarm signal arrives at an importune moment, the system call is interrupted and only a part of the buffer is written out. Most current *Stdio* implementations will note that an incomplete write had happened but nothing is done about the part of data not yet written out. The result is a loss of data.

- *Inefficiency*: *Stdio* enforces the notion that the application has to provide a buffer (different from the stream buffer) to transfer data in and out of streams. This causes unnecessary data copying in many common cases. In addition, it makes it hard to get data whose size may not be known a priori to a call, for example, a line of text. There are recent proprosals to add a new function `fgetline()` precisely for this purpose. Even if this is done, it is a kludge to get around the fundamental problem of how to safely access stream data without unnecessary data copying.

  In many versions of *Stdio*, an I/O operation entails reading or copying data into the internal stream buffer before transferring it to the final destination. For reading or writing an amount of data that

exceeds the buffer size, this is wasteful. For example, a call like `fread(buf,10,BUFSIZ,stdin)` may imply multiple invocations of the `read()` system call.

- *Inadequacy:* Parts of *Stdio* simply fall short of what the interface promises. For example, the `fread()` and `fwrite()` functions promise I/O of objects which may have large sizes. Since the system calls `read()` and `write()` only promise I/O of byte streams, this is a promise that is hard to keep. For example, if a structure of size, say 100, is written out to a pipe but the `write()` system call is interrupted after the first 10 bytes is copied, `fwrite()` can only report that no structure has been written out. This is misinformation at best.

  Some implementations of *Stdio* are also inadequate resulting in unnecessary and complex interface requirement. For example, *Stdio* allows streams to be opened for both read and write. However, applications are not allowed to arbitrarily mix input and output operations. Instead, extra `fseek()` calls must be inserted before switching mode. This requirement results because current implementations of *Stdio* do not have a way to know what mode the stream is in. From an application's point of view, if a stream is passed as an argument to some function, since there is no way to know what mode it is in, the application may have to use a defensive measure causing many unnecessary calls to `fseek()`.

  Buffer synchronization for streams sharing the same file is a good source of problems for *Stdio* programmers. The following code fragment can cause unexpected results:

```
printf("Please type something:");
fgets(buf,sizeof(buf),stdin);
printf("You just typed: %s",buf);
```

  Because of buffering, the user will not see the prompt until s/he has typed a line of input. Then both the prompt and the typed text will be printed. Strictly speaking, this is not an error on *Stdio* part. It just lacks a facility that would allow the above "natural" piece of code to work as expected (see section 3.4). A sharp reader may bring up `setbuf(stdout,NULL)` or `fflush(stsdout)` here but s/he should be forewarned at least of the efficiency cost incurred in the former case and the inconvenience of the latter. This is not to mention other more complex buffer synchronization situations such as two streams writing to the same file. Can the output be guaranteed to serialize logically?

- *Insecurity:* Parts of the interface of *Stdio* are notoriously insecure. For example, the *gets()* and *sprintf()* functions allow overwriting data space that follows a buffer. Below is an example of unsafe coding style possible with *Stdio*. The existence of such a coding style in standard system utilities was rumored to have been put to good use in at least a few system break-in techniques.

```
char buf[1];
sprintf(buf,"1234");
```

  More benign but equally troublesome for applications is the possibility of concurrent access of streams due to signal handling. In such cases, the behavior of the stream is unspecified since internal stream pointers may be in a bad state during a concurrent access.

- *Inconsistency:* An annoying problem with the interface design of *Stdio* is the positioning of the stream argument in a function call. For example, `fseek()` and `setbuf()` require the stream argument to come first while `fread()` and `fwrite()` require it to come last. Though this is not a fatal flaw, inconsistency of this type steepens the learning curve for programmers. Some of us, after many years of programming with *Stdio*, still often have to open the manual to find the correct function calling sequences.

- *Incompleteness:* This is an open area where everyone's wish list can be inserted. However, *Stdio* does lack a number of fundamental features. Most important is a lack of some way for applications to handle exceptions such as the end-of-file condition or interrupted system calls. More along the efficiency line, there is no safe method to directly access the internal buffer of a stream. The inefficiency resulting from this have forced many applications to invent their own buffering scheme.

Except for `sprintf()` and `sscanf()`, the library is closedly tied to files. There is no capability to manipulate arbitrary data streams so the convenience of the I/O operations are lost for memory manipulation. This is a lack recognized by providers of the *C++ Iostream* package [St].

3. The Sfio Library

3.1 Design Considerations

The *Sfio* library was built to correct the problems of *Stdio* and to simplify our life as programmers. Interface design considerations concluded for us that a rewrite of *Stdio* just wouldn't do so we started fresh. Below are some of the considerations that went into the design of *Sfio*.

- *Efficiency*: A package as basic as *Sfio* must be efficient in its implementation. The I/O primitives in *Sfio* are designed to avoid any unnecessary data copying. New and efficient algorithms are used when appropriate. For example, the *Sfio* `sfprintf()` and `sfscanf()` functions use new data conversion algorithms that are much faster than their counterparts in *Stdio*.

  In modern UNIX systems, there are often multiple ways to do I/O with different trade-offs. For example, on SUN OS and System V Release 4, it is possible to use memory mapping to access file data. *Sfio* takes advantage of these facilities where appropriate to speed up thoughput.

  Beyond internal efficiency, an I/O package like *Sfio* must also provide primitives that allow an application to be efficient in its I/O manipulations. An example is `sfpeek()` which allows an application safe access to the internal buffer of a stream. We have rewritten many standard system utilities based on this primitive with good performance improvement. It is worth noting that because of efficiency problems with *Stdio*, popular tools such as *cat* or *cp* were often rewritten using raw system calls. However, by going to lower level system calls, these implementations cannot take advantage of new efficient features such as memory mapping without further recoding. Our implementations of these tools based on *Sfio* achieve the same or better efficiency.

- *Consistency*: All *Sfio* constants begin with the prefix `SF_` and all functions begin with the prefix `sf`. Where applicable, *Sfio* functions simply emulate their system call counterparts. For example, the `sfread()` function is called as `sfread(stream,buf,size)`. This example also shows that the stream argument always positions first in any function call. Then, if required, the second argument is a buffer and the third would be the buffer size.

  All streams are handled in the same way by the application. There is no difference between string (memory only) and file streams. This is used by `sftmp()`, a function to create streams for storing and retrieving temporary data. For efficiency, such a temporary stream starts out being a string stream with a buffer of some size decidable by the application. A stream discipline is set up so that if this memory area is exhausted, a true temporary file will be created. From the application's point of view such a temporary stream simply appears as if it is always a file.

- *Orthogonality:* All stream-applying primitives in *Sfio* are orthogonal so they can be used in arbitrary orders. For example, the `sfsetbuf()` functions can be called any time to change the buffer of a stream, not only when the stream is just opened. This is important for *Sfio* since it supports string streams that correspond to in-memory data buffers. In the same vein, there is no restriction on the order of I/O operations on streams that are opened for both read and write.

- *Integrity:* Except for convoluted escape hatches (e.g., `longjmp()`), the library guarantees that internal stream buffers maintain their integrity when exceptions occur. For example, if a `write()` system call is interrupted, no data will be lost (unless, of course, if `write()` itself loses it). An application can also set alternative disciplines to handle such exceptions. Streams that are opened for read only or write only will disallow the opposite operations. Finally, during an operation that modifies a stream, the stream is locked to prevent other concurrent accesses to the same stream.

- *Robustness:* We have mentioned the use of locks to prevent concurrent stream accesses. The library also prevents most unsafe operations that may cause buffer overflow. Except for string scanning patterns such as `%s` or `%[]` in `sfscanf()`, there is no way to specify a buffer without an accompanying size. The only reason for saving the current semantics of these patterns despite their

insecurity is a concession for portability of the legion of existing applications. We do provide alternative patterns, `%S`, `%C` and `%{}`, which require the accompanying buffer sizes.

For string inputs, `sfpeek()` can be used in a line-buffering read stream to read lines shorter than the buffer size. Unbounded lines can also be read via `sfgets(stream,NULL,SF_UNBOUND)` which reads such a line into some internally allocated memory area.

- *Minimality:* As a general rule, we avoid providing functions unless they do something that cannot be done outside of *Sfio* without some loss. For example, there are no functions corresponding to the `getchar()`, `putchar()` family since they only provide short-hand notations to functions applying to the standard output stream. There are border cases where this rule is relaxed somewhat. For example, `sfsprintf()` and `sfsscanf()` are provided even though the respective input/output string could be opened for I/O as a string stream. Their popularity and the saving of creating a stream structure is worth their existence.

  At the implementation level, we try to minimize the amount of code that gets pulled in with normal compilation. For example, even though stacked streams can only be closed as a unit, the code for stack manipulation is never pulled in unless stacking is used in the application.

- *Extensibility:* No library provider can anticipate all possible needs that an application may want from the package. Thus, we have made a stream parametrizable via the use of disciplines. A discipline redefines the system calls `read()`, `write()`, and `lseek()`, and provides a function to handle exceptions. Different disciplines can be stacked on one another to make a sequence of filters. For example, the *pack* and *unpack* utilities have been rewritten based on a discipline that provides a virtual plaintext view of writing and reading packed files. The beauty in such a scheme is that the same discipline can be reused in other applications for accessing the same type of data.

3.2 Creating and Deleting Streams

Sfio defines a stream type called `Sfile_t` and a collection of functions to create, manipulate, and destroy streams. A stream can be created from a file descriptor or memory. It is common to call a memory stream a string stream. Streams can be created for reading, writing or both and its I/O mode can be set for byte-oriented or line-oriented.

The primary mechanism for creating a stream is `sfnew()`:

```
sfnew(Sfile_t* f, char* buf, int size, int fd, int type)
```

`f` is a stream pointer. If it is `NULL` or not closable, a new stream is created; otherwise, `f` is a stream to be closed and reused in making a new stream. `buf` and `size` refer to a buffer and its size when `size` is positive. If `size` is 0, the stream is unbuffered and if `size` is negative, the actual buffer will be allocated by the system. `fd`, refers to a file descriptor when the stream models a file stream and is ignored otherwise. `type` is a bit pattern defining the type of stream. It includes the bits:

`SF_STRING`: This bit indicates that the stream is an array of bytes in memory, not a file. If the stream is opened for reading, `buf` and `size` define the data and its extent.

`SF_READ` and `SF_WRITE`: These bits specify read and write mode.

`SF_LINE`: This bit turns on the line mode. For write streams, this means flushing the buffer whenever the new-line character is output. For read streams, the `sfpeek()` function (below) will return data corresponding to a line ending in a new-line character.

`SF_SHARE`: This bit indicates that the stream corresponds to a file descriptor that may be shared elsewhere by a different stream or a different process. The file position will be reset to its logical location before a call to `read()` or `write()`.

`SF_APPEND`: This bit indicates that the stream is a file opened for append. On systems where this mode is not available, a seek to the end of the file will be performed before a call to `write()`.

Stream objects can also be created from higher level functions: `sfopen()`, `sfdopen()`, and `sfpopen()`. `sfopen()` is similar to the `freopen()` function in *Stdio*. However, when the first

argument is `NULL`, it creates a new stream. In addition, if the third argument is `"s"` or `"s+"`, the second argument, instead of being a file name, is a string to be opened for read or read and write. `sfpopen()` is similar to `popen()` in *Stdio*:

```
sfpopen(const char* cmd, const char* type, Sfile_t** fcomp)
```

Here, `cmd` is a command connected to the application via pipes. `type` determines the type of connection. It can be `"r"`, `"r+"`, `"w"`, or `"w+"`. If the + modifier is used, a companion stream of the opposite type is created and returned in `*fcomp`. If two streams are created, they will be pooled with the type `SF_WRITE` (see section 3.4).

The function `sfclose()` is the only method to close a stream. If a stream corresponds to a process opened by `sfpopen()`, its companion stream, if any, is also closed. Finally, `sfclose(NULL)` can be used to close all streams.

3.3 Reading and Writing Streams

All legal I/O operations can be performed on a stream in arbitrary order. There are five ways to do I/O on streams:

- *Byte or number oriented*: The function `sfgetc(f)` returns the next byte from the stream `f`. If there are no more data, it returns `SF_EOF`. The function `sfungetc(f,c)` can be used to push back a byte into the stream. Unlike *Stdio*, there is no limit to the number of bytes that can be pushed back into a stream. The inverse of `sfgetc()` is `sfputc()` which writes a byte into the stream. Other functions, `sfgetl()`, `sfputl()`, `sfgetu()`, `sfputu()`, `sfgetd()`, and `sfputd()`, are available to read and write integral and floating point values in a portable format. The coding format is portable as long as the bit order within a byte is the same across the relevant machines and if two corresponding types have the same size. The coding for floating point values rely on the system provided functions `frexp()` and `ldexp()`. Most programs manipulate only small values; our coding method takes advantage of this and uses a minimal number of bytes. This saves disk space when storing large amounts of data. The cost of encoding and decoding data is usually well paid for by the time saved in accessing disk data.

- *String-oriented*: The function `sfgets(f,buf,n)` reads up to and including the new-line character into the buffer pointed to by `buf`. At most `n-1` characters is read and a 0 byte is appended. If `buf` is `NULL` or `n` is non-positive, the data is read into an internal buffer which will be allocated as necessary. Thus, in this case, there is no hard limit on the length of the input string. After a call to `sfgets()`, the function `sfslen()` can be called to get the length of the input string.

- *Block-oriented*: The function `sfread(f,buf,n)` tries to read `n` bytes from `f` into the buffer `buf`. It returns the number of bytes actually read. Similarly, `sfwrite(f,buf,n)` writes `n` bytes to the stream `f`. The function `sfmove(fr,fw,n,rs)` is used to move data from the stream `fr` to the stream `fw`. If `n` is non-negative, it indicates the number of units to move; otherwise all data in the stream is moved. The movement unit is either a byte or a record delineated by the record separator `rs` when `rs` is non-negative. The below example skips `10` lines from the standard input, then moves the next `10` lines to the standard output.

  ```
  sfmove(sfstdin,NULL,10,'\n');
  sfmove(sfstdin,sfstdout,10,'\n');
  ```

- *Formatted I/O*: The functions `sfscanf()` and `sfprintf()` provides ways to read and write data that are formatted. These functions are similar to the `fscanf()` and `fprintf()` functions of *Stdio* though we have provided a few extensions so an application can interpret unknown patterns or provide a different way to get or assign formatted arguments. See the appendix for details.

- *Direct stream buffer access*: This is the fastest method for performing I/O using *Sfio*. It is done via the function `sfpeek(f,bufp)`. If `f` is a read stream, `sfpeek()` fills the buffer if it is empty, then return in `*bufp` the pointer to the beginning of available data. The return value of `sfpeek()` indicates the extent of the data. If the stream is in `SF_LINE` mode, `sfpeek()` will make sure, without exceeding the buffer size, that the buffer contains a line ended by new-line and

return the length of the line. If `f` is a write stream, `sfpeek()` flushes the buffer if it is full, then returns in `*bufp` the pointer to the start of the buffer area available to write. In this case, the return value of `sfpeek()` indicates the size of the writable portion of the buffer. Note that `sfpeek()` does not change the current stream location! That must be done via a normal `sfread()` or `sfwrite()` call. Below is an example of using `sfpeek()` to get access to a write buffer, put some data into it, then advance the write pointer past the data. This example works because `sfwrite()` will notice that `buf` points to the same position as the next write position in the stream buffer and simply advances the write position without doing any data copying.

```
p = sfpeek(f,&buf);
n = process(buf,p);
sfwrite(f,buf,n);
```

3.4 Stream Synchonization

When a stream corresponds to a file, *Sfio* either buffers data or use memory mapping to reduce the number of `read()` and `write()` system calls. This means that the logical seek location of a stream may not correspond to the seek location of the underlying file. The `sfsync(f)` function can be used to cause the two seek locations to synchronize. For a write stream, any buffered data will be output. For a read stream, the file seek location may be rolled back as necessary and if possible. The call `sfsync(NULL)` synchronizes all streams.

Streams can be grouped together into auto-synchronizable pools using the `sfpool(f,poolf,type)` function. If `poolf` is `NULL`, `f` is taken out of its current pool, if any. Otherwise, `f` is pooled with `poolf`. In each pool, the most recently accessed stream is at the head of the pool. When a new stream is accessed, if the type of the current head stream matches `type`, it will be synchronized with `sfsync()` before the new stream is moved to the head of the pool. The `type` argument of `sfpool()` can be any non-empty combination of `SF_READ` and `SF_WRITE`. Below is an example of pooling the standard input and output streams so that the standard output is always flushed before reading from the standard input:

```
sfpool(sfstdin,sfstdout,SF_WRITE)
```

The equivalence of the above construct is, in fact, done automatically by the library on initialization of `sfstdin`, `sfstdout` and `sfstderr` if they are unseekable streams such as terminals or pipes. This solves the buffer flushing problem mentioned in section 2. Another example of `sfpool()` is to pool streams sharing the same file. This helps preserving the order of I/O operations when they are mixed among streams. For example, in *Sfio*, physical outputs to the default streams `sfstdout` and `sfstderr` are guaranteed to be in the order that the output operations are performed.

3.5 Stream Stacking

There are many applications in which it is desirable to insert a stream of data at some point of input. For example, the C preprocessor routinely needs to insert a new macro definition or an include file. The `sfstack(base,f)` function inserts a new stream `f` on top of the stack referred to by the `base` stream. All I/O operations are requested via the `base` stream but actually performed on the top stream. A top stream will be popped and closed on end-of-file. The stack can also be popped by specifying `NULL` for the second argument of `sfstack()`. Below is an example of using stream stacking. The function `process()` checks to see if a line of text defines an include file and returns its name.

```
while((s = sfgets(sfstdin,NULL,SF_UNBOUND)) != NULL)
{   char *include = process(s);
    if(include)
    {   Sfile_t *f = sfopen(NULL,include,"r");
        if(f)
            sfstack(sfstdin,f);
    }
}
```

In the next section, we give an example showing how a discipline can be used in conjunction with stacking to catch the end-of-file event of the inserted file and reset certain state information such as the current file name and line number.

3.6 Stream Discipline

The function `sfdisc(f,disc)` pushes a new I/O discipline onto the stream `f`. A discipline specifies functions to replace the system calls `read()`, `write()`, and `lseek()`, and to handle exceptions. The argument `disc` is either `NULL` to pop the discipline stack or a pointer to a `Sfdisc_t` structure. This structure contains four publically visible fields: `(*readf)()`, `(*writef)()`, `(*seekf)()`, and `(*exceptf)()`. The first three fields specify alternative I/O functions. If one of these functions is `NULL`, it is inherited from an earlier discipline on the discipline stack. For completeness, the bottom of the stack of a file stream always contains the system call discipline. The fourth field, `(*exceptf)()`, defines an exception handler.

A discipline I/O function, `(*readf)()`, `(*writef)()`, or `(*seekf)()`, is called with 4 arguments. The first argument is the stream pointer. The second and third arguments correspond to the second and third arguments of the respected system call. The fourth argument is the discipline structure, `disc`, itself. Note that since a discipline function is invoked during a stream I/O operation, the stream remains locked during its execution. A discipline function should avoid using I/O system calls directly and use `sfrd()`, `sfwr()` and `sfsk()`. These functions invoke lower level discipline functions and ensure proper exception handling.

The exception function, `(*exceptf)()` is called when a read or write exception happens, when a stream is being closed, or when the discipline is being reset. A read or write exception occurs when the discipline I/O function returns a zero or negative value. During a call to `(*exceptf)()`, if it is at the top of the discipline stack, the stream will be opened for general operations. `(*exceptf)()` is called as `(*exceptf)(f,type,disc)`. Here, `type` is: `SF_DPOP` when the discipline is about to be popped of the discipline stack, `SF_DPUSH` when the discipline is about to be pushed down in the stack, `SF_CLOSE` when the stream is being closed, `SF_READ` when an exception happens during a read operation, and `SF_WRITE` when an exception happens during a write operation. The current *Sfio* function will examine the return value of `(*exceptf)()` for further actions: negative for immediate return, zero for executing default actions associated with the exception, and positive for resuming execution as if no exception happened.

Here is an example of using a discipline to copy data from the standard input to the standard output where all upper case characters are translated to lower case.

```
lower(Sfile_t* f, char* buf, int n, Sfdisc_t* disc)
{   int c;
    n = sfrd(f,buf,n,disc);
    for(c = 0; c < n; ++c)
            buf[c] = tolower(buf[c]);
    return n;
}
...
Sfdisc_t Disc = { lower, NULL, NULL, NULL, NULL };
sfdisc(sfstdin,&Disc);
sfmove(sfstdin,sfstdout,SF_UNBOUND,-1);
```

As usual, there are different solutions with trade-offs in ease of coding, ease of understanding, reusability, and efficiency. The above solution is attractive because it isolates the translation act into a single, easily recognizable and reusable function. The cost of an extra function call per buffer filling is minimal since it typically occurs only once per 8Kb of data.

In practice, most stream disciplines require state information. This can be done by defining application-specific discipline structures. For example, stacked streams frequently require the save and restore of line numbers and file names. A discipline structure for a stacked stream can be defined as:

```
typedef struct  _stk_disc
{   Sfdisc_t    disc;
    int         line;
    char        *file;
} Stk_disc_t;
```

The discipline exception function will restore these values when the stream is about to be closed:

```
stk_except(Sfile_t* f, int type, Sfdisc_t* disc)
{   if(type == SF_CLOSE)
    {   Line = ((Stk_disc_t*)disc)->line;
        File = ((Stk_disc_t*)disc)->file;
    }
    return 0;
}
```

The application code needs only set this discipline before stacking the new stream:

```
Stk_disc_t *stk_disc = (Stk_disc_t*)calloc(1,sizeof(Stk_disc_t));
stk_disc->disc.exceptf = stk_except;
stk_disc->line = Line;
stk_disc->file = File;
sfdisc(f,(Sfdisc_t*)stk_disc);
sfstack(sfstdin,f);
```

3.7 Extensions to sfprintf() and sfscanf()

The formatted I/O routines `sfprintf()` and `sfscanf()` have a few extensions for added security and to allow an application to tailor their processing. The string scanning patterns `%C`, `%lc`, `%S`, `%ls`, `%{}` and and `%l[]` are analogues of `%c`, `%s`, and `%[]` but they require a buffer size. Three additional customizing patterns are supported: `%&`, `%@` and `%:`. The `%&` pattern indicates that the next argument in the variable argument list is the address of a function that is used to interpret patterns not already defined by the respective utility. The `%@` pattern indicates that the next argument is the address of a function to get arguments in the case of `sfprintf()` and to assign values in the case of `sfscanf()`. This is useful, for example, to write an interpreter version of `printf()` where the arguments are unknown at compile time. Note that both extension functions are local to an invocation of `sfprintf()` or `sfscanf()`. This allows recursive calls of these functions (via the extension functions) to define different types of processing. Finally, the pattern `%:` indicates that the next two arguments are a pattern string and a corresponding variable argument list to be inserted. The processing continues with the new pattern string until it is exhausted, then resumes with the earlier string.

4. Stdio Compatibility

To ease the transition for programs currently based on *Stdio*, we wrote both source and binary compatibility packages. At the source level, the `FILE` type is simply redefined to be `Sfile_t`. Most *Stdio* functions and constants can be emulated by macro definitions to their *Sfio* counterparts. This is done in an emulation `stdio.h` header file. A few functions such as `printf()` and `scanf()` that accepts variable argument lists are reimplemented based on the *Sfio* versions.

Binary compatibility means that the library must work with code already compiled with the standard *Stdio* header file. In general, this is unsolvable because an *Stdio* implementation may define the `FILE` structure in unpredictable ways. A working solution for most *Stdio* implementations is to write a program to generate from the header file `stdio.h` an emulation structure that contains the fields `_cnt`, `_ptr`, `_flag` and `_file` in the exact locations that they appear in the `FILE` structure. This keeps correct the embedded pointers in macro functions such as `getc()` or `putc()`. Functions are written to provide one-to-one mappings between the set of `FILE` emulation structures to the set of `Sfile_t` structures. Then, each *Stdio* function is reimplemented to map a given `FILE` structure to its `Sfile_t` counterpart before calling the appropriate *Sfio* function. The reimplemented functions maintain the `_flag` field up to date with respect to I/O exceptions. Finally, by initializing the `_cnt`

field to be 0, the macro functions `putc()` and `getc()` are trapped to call `_filbuf` and `_flsbuf`, internal *Stdio* functions to fill and flush buffers. Reimplementing these functions to call their *Sfio* counterparts and to set `_cnt` and `_ptr` to the appropriate values completes the job.

5. Performance

5.1 I/O Benchmark Results

To measure relative performances of *Sfio* and comparable packages, *Stdio* and *Fio*, we wrote a benchmark program that exercises the basic I/O functions. The program is based on the *Stdio* interface so that we can measure the performance of *Stdio*. To measure the performance of *Sfio*, we simply recompiled the program using the source *Stdio* emulation package. Benchmark data were obtained under a variety of environments. To reduce variance, each set of data was an average of five different runs at night time. In the data tables, the first column shows the functions being tested. The second column shows the amount of data processed in units of lines or kilobytes. or number of seeks in the case of *seek+rw*. The remaining columns partitions into set of three each. In each set of columns, the first shows the amount of *cpu* time, the second *system* time, and the third throughput rate which is computed by dividing the total amount of data in K-bytes by the total cpu and system time. The first three tests show results of block I/O where each block is of size 8Kb. The *revrd* test reads blocks in reverse order, i.e., starting from the bottom of the file. The next three tests show results of block I/O with blocks of size 757, a number that does not divide 8K. The tests *copy&rw* and *sfmove* copy a large file by either `fread()` and `fwrite()` or the `sfmove()` primitive of *Sfio*. The test *seek+rw* performs a sequence of seeking to a random location, reading a block of size 8K, then copying that block to location 0. The tests *fputc* and *fgetc* perform byte I/O. The *fputs*, *fgets* and `revgets` tests perform line I/O. *revgets* reads lines in reverse order. Both the `printf()` and `scanf()` tests exercise all common conversion modes: `%c`, `%d`, `%o`, `%x`, `%f`, `%e` and `%s`.

The data will be presented in four tables. Table 1 shows benchmark results on a two-processor Solbourne running SUN OS4.0, comparing *Sfio*, *binary* (the *Stdio* binary emulation package), and *Stdio*. In this version, *Sfio* uses memory mapping for all read streams. This is why `sfmove()` is twice as fast as copying data by block reads and block writes. The functions `sfprintf()` and `sfscanf()` are much faster than their counterparts in *Stdio* due to new data conversion algorithms which, among other improvements, avoid division and multiplication which are expensive on a SPARC machine.

Table 2 compares *Sfio* and *Stdio* on a VAX 8650 running 4.3BSD. Many parts of the native *Stdio* package are implemented in assembly code, especially `fputs()`, `fgets()` and `_doprnt()`, the guts of `fprintf()`. In the VAX version of *Sfio*, `sfprintf()` is written entirely in C but `sfgets()` and `sfputs()` contain a few `asm()`s to access hardware instructions for block search and block copy.

Table 3 compares *Sfio* and *Stdio* on an Intel 386 machine running System V Release 3.2. Since this machine has limited disk space, we usesd smaller data sets.

Table 4 compares *Sfio*, *Stdio* and Hume's *Fio*. This is done on a MIPS machine running UMIPS. Again because of limited disk space, we used smaller data sets. As *Fio* does not have the same interface as *Stdio*, a few emulation macros were written to map the needed functions. The poor performance of *Fio* analogues of `fputc()` and `fgetc()` is due to their implementation as subroutines.

5.2 Reimplemented System Utilities

A goal of *Sfio* is to be sufficiently efficient so that applications can be based on it without fear of performance loss. To test this, we rewrote several standard UNIX utilities including *ksh*, *cat*, *wc*, *cut*, *pack*, and *unpack*. *Ksh* is a complex program that exercises virtually all aspects of *Sfio*. *Cat* is an I/O bound program and, to a lesser extent, so is *wc*. *Cut* is a non-trivial example where I/O is still likely to matter. *Pack* and *unpack* show how the Huffman coding can be implemented as a discipline. For example, *unpack* entails opening a packed file, setting the unpacking discipline, then transferring data as if it is in plain text.

		Sfio			*binary*			*Stdio*		
test	size	user	sys	Kb/s	user	sys	Kb/s	user	sys	Kb/s
fwrite	10000K	0.03	2.04	4813	0.09	2.13	4504	0.05	2.03	4796
fread	10000K	1.00	0.62	6163	1.07	0.66	5755	0.74	1.43	4608
revrd	10000K	0.31	1.62	5188	0.35	1.54	5298	0.79	1.39	4597
fw757	10000K	1.12	1.97	3236	1.25	2.06	3016	0.96	1.91	3481
fr757	10000K	1.20	0.58	5610	1.36	0.66	4944	0.96	1.38	4278
rev757	10000K	1.25	2.66	2560	1.71	2.58	2325	1.21	15.28	606
copy&rw	10000K	1.13	2.88	2496	1.25	2.94	2389	0.88	3.87	2105
sfmove	10000K	0.01	1.90	5228						
seek+rw	2000S	0.83	6.35	2268	0.90	6.07	2315	1.52	5.87	2185
fputc	5000K	3.92	1.14	987	4.29	1.08	929	4.18	1.14	940
fgetc	5000K	3.78	0.44	1184	4.02	0.23	1175	3.93	0.79	1059
fputs	50000L	2.15	1.03	1538	2.63	1.18	1285	2.07	1.08	1549
fgets	50000L	2.21	0.25	1981	2.58	0.33	1681	2.05	0.70	1783
revgets	50000L	2.56	1.25	1309	3.49	1.28	1028	4.15	36.59	122
fprintf	50000L	6.06	1.00	506	6.81	0.97	458	16.71	1.31	196
fscanf	50000L	6.56	0.29	518	7.47	0.41	454	17.90	1.00	187

TABLE 1. Solbourne and SUN OS4.0

		Sfio			*Stdio*		
test	size	user	sys	Kb/s	user	sys	Kb/s
fwrite	10000K	0.11	3.90	2496	1.11	4.09	1924
fread	10000K	0.11	1.78	5284	0.79	1.71	3996
revrd	10000K	0.18	1.88	4854	1.00	2.00	3336
fw757	10000K	0.97	4.32	1893	0.83	4.42	1904
fr757	10000K	1.07	1.78	3514	0.80	1.72	3976
rev757	10000K	1.47	4.98	1550	2.15	19.02	472
copy&rw	10000K	0.26	4.97	1911	1.87	5.19	1416
sfmove	10000K	0.04	4.87	2035			
seek+rw	2000S	1.23	7.41	1851	3.02	7.87	1469
fputc	5000K	8.84	1.93	464	10.18	1.92	413
fgetc	5000K	6.74	1.07	640	8.00	1.04	553
fputs	50000L	1.33	2.15	1405	1.15	2.21	1456
fgets	50000L	1.21	0.93	2290	0.93	0.90	2659
revgets	50000L	2.29	2.30	1063	5.12	68.17	66
fprintf	50000L	18.86	1.49	174	22.28	1.67	147
fscanf	50000L	25.34	0.93	134	42.11	1.18	81

TABLE 2. VAX 8650 and 4.3BSD

Table 5 compares different versions of *cat*, *wc*, *cut*, *pack* and *unpack*. The data were obtained on a two-processor Solbourne running SUN OS4.0. Each program is run with two different datasets, one of size 50000 bytes and the other 5 million bytes. The data for each utility will be presented in two successive rows, the smaller dataset first, then the larger one. Where it is applicable, we show data of three different versions of the tool, SUN OS, System V, and our implementation. Note that *cut*, *pack* and *unpack* are the same on SUN OS and System V.

The data shows that our versions of the standard utilities perform at the same level or better than the other versions. The *cat* test shows that, even with the direct use of `read()` and `write()` for data transfer, the System V *cat* still lost in system time to its SUN-OS and *Sfio* counterparts because the latter use memory mapping. The data for *wc* shows another aspect of efficiency at the library level. In

		Sfio			*Stdio*		
test	size	user	sys	Kb/s	user	sys	Kb/s
fwrite	1000K	0.02	2.91	340	0.02	2.87	346
fread	1000K	0.02	3.26	305	0.24	3.87	242
revrd	1000K	0.03	2.91	340	0.24	3.26	285
fw757	1000K	0.31	2.84	317	0.41	3.16	279
fr757	1000K	0.39	3.40	263	0.43	3.67	243
rev757	1000K	0.47	5.58	165	0.42	5.44	170
copy&rw	1000K	0.03	6.37	156	0.25	6.65	144
sfmove	1000K	0.01	6.60	151			
seek+rw	500S	0.46	7.66	491	1.32	11.58	309
fputc	500K	3.81	1.50	94	3.73	1.75	91
fgetc	500K	3.44	1.70	97	3.16	1.86	99
fputs	10000L	2.72	2.60	185	2.66	3.09	169
fgets	10000L	2.58	3.28	167	2.58	3.83	152
revgets	10000L	2.32	4.18	149	4.20	32.74	26
fprintf	10000L	32.96	2.01	19	32.66	2.38	19
fscanf	10000L	30.60	2.56	20	38.60	2.63	16

TABLE 3. Intel 386 and System V Release 3.2

		Sfio			*Stdio*			*Fio*		
test	size	user	sys	Kb/s	user	sys	Kb/s	user	sys	Kb/s
fwrite	1000K	0.01	1.91	521	14.26	1.75	62	0.19	2.01	456
fread	1000K	0.01	0.91	1086	14.12	0.91	66	0.01	1.01	978
revrd	1000K	0.02	0.84	1152	13.84	0.92	67	0.04	0.79	1187
fw757	1000K	0.45	1.75	453	14.26	1.70	62	0.45	1.83	437
fr757	1000K	0.43	0.84	787	14.05	1.18	65	0.19	2.54	365
rev757	1000K	0.57	2.17	364	14.04	2.39	60	0.28	2.84	320
copy&rw	1000K	0.05	2.62	374	28.02	2.61	32	0.18	2.71	346
sfmove	1000K	0.01	2.65	376						
seek+rw	500S	0.74	5.79	612	111.92	6.53	33	0.92	6.23	559
fputc	500K	5.07	1.02	82	4.88	0.93	86	20.18	0.98	23
fgetc	500K	4.30	0.48	104	4.44	0.46	101	18.52	0.45	26
fputs	10000L	5.15	1.74	141	9.74	1.79	84	5.49	1.82	133
fgets	10000L	5.04	0.95	162	11.17	0.94	80	4.96	1.11	160
revgets	10000L	6.16	2.06	118	12.09	5.26	55	6.69	38.17	21
fprintf	10000L	20.74	1.18	31	60.08	1.40	11	52.33	1.29	13
fscanf	10000L	27.06	0.75	24	43.85	0.74	15			

TABLE 4. MIPS and UMIPS

this case, the SUN-OS version of *wc* consumes much more cpu time than System V version because it uses `getc()` for input while the System V version uses `fread()`. The *Sfio* version improves on the System V version via the use of `sfpeek()` and a new algorithm for detecting word and line boundaries. The big performance improvement in *unpack* is due in large part to a new unpacking algorithm which makes use of `sfpeek()` and discipline. The benefit of coding *unpack* as a discipline is that the discipline can be reused directly in other contexts.

6. Conclusions

We presented *Sfio*, a new library for stream input/output that can be used as a replacement for *Stdio*, the standard I/O library for C. A number of deficiencies in *Stdio* were described. We discussed how *Sfio*

	SUN-OS		*System V*		*Sfio*	
command	user	sys	user	sys	user	sys
cat	0.01	0.08	0.02	0.09	0.01	0.11
	0.01	0.09	0.05	1.08	0.03	0.17
wc	0.10	0.09	0.04	0.08	0.04	0.10
	5.72	0.75	2.55	0.69	1.57	0.68
cut	0.11	0.11	0.11	0.11	0.05	0.10
	9.69	1.21	9.69	1.21	3.79	1.30
pack	0.11	0.16	0.11	0.16	0.09	0.11
	9.92	3.76	9.92	3.76	6.87	1.40
unpack	0.24	0.13	0.24	0.13	0.10	0.10
	19.65	2.96	19.65	2.96	6.90	1.59

TABLE 5. Reimplemented System Commands

avoids such deficiencies. Performance results from our own benchmark program and a few reimplemented utilities were shown The data shows that *Sfio* performs as well as or better than *Stdio* on popular hardware platforms running different UNIX versions. This is nice especially since many versions of *Stdio* that we compare *Sfio* against have been hand-tuned in assembly code. Standard utilities gain substantial efficiency when reimplemented based on *Sfio*. Despite many new features, the total size of the library is under 50K on a SPARC1+ workstation.

Beyond a being a better replacement for *Stdio*, *Sfio* introduces a number of new abstractions. Streams have been generalized to represent both files and in-core memory areas. Fast access to internal stream buffers is provided so that applications should never have to revert to bare system calls for efficiency. New methods are provided to handle synchronization of collections of streams, build new virtual streams by stacking and change stream I/O disciplines. Stream disciplines allow applications to tune the handling of exceptions and to change or augment the processing power of the underlying system calls. Many routine data processing tasks can be modeled and implemented by appropriate disciplines resulting in better code organization and code reuse. In our experiences, the combination of performance and new features makes *Sfio* a powerful tool that enhances our ability to write programs.

Acknowledgement

Glenn Fowler helped firm up the design and implementation of *Sfio*. Griff Smith contributed the main idea and code behind the new decimal conversion algorithm for `sfprintf()`. Finally, thanks are due to Andrew Hume and Griff Smith who helped to improve the presentation of the paper.

References

[AN] ANSI x3.159-1989, *American National Standard for Information Systems - Programming Language - C*, Amer. Nat. Sta. Ins., 1990.

[Hu] A. Hume, *A Tale of Two Greps*, Soft. Prac. & Exp., v.18, pp.1063-1072, 1988.

[KR] B. Kernighan & D. Ritchie, *The C Programming Language*, Prentice Hall, 1978.

[St] B. Strousup, *The C++ Programming Language*, Addison-Wesley, 1986.

NAME

sfio – safe/fast string/file input/output

SYNOPSIS

```
#include    <sfio.h>

#define ulong           unsigned long

Sfile_t*    sfnew(Sfile_t* f, char* buf, int size, int fd, int flags);
Sfile_t*    sfopen(Sfile_t* f, const char* string, const char* mode);
Sfile_t*    sfdopen(int fd, const char* mode);
Sfile_t*    sfpopen(const char* cmd, const char* mode, Sfile_t** fcomp);
Sfile_t*    sfstack(Sfile_t* base, Sfile_t* top);
Sfile_t*    sfpushed(Sfile_t* f);
Sfile_t*    sftmp(int size);

Sfdisc_t*   sfdisc(Sfile_t* f, Sfdisc_t* disc);
int         sfrd(Sfile_t* f, char* buf, int n, Sfdisc_t* disc);
int         sfwr(Sfile_t* f, char* buf, int n, Sfdisc_t* disc);
long        sfsk(Sfile_t* f, long addr, int offset, Sfdisc_t* disc);

int         sfclose(Sfile_t* f);
int         sfsync(Sfile_t* f);
int         sfpool(Sfile_t* f, Sfile_t* poolf, int mode);

int         sfpeek(Sfile_t* f, char** bufp);

int         sfgetc(Sfile_t* f);
int         sfungetc(Sfile_t* f, int c);
ulong       sfgetu(Sfile_t* f);
long        sfgetl(Sfile_t* f);
double      sfgetd(Sfile_t* f);
char*       sfgets(Sfile_t* f, char* buf, int size);
int         sfread(Sfile_t* f, char* buf, int n);
int         sfscanf(Sfile_t* f, const char* format, ...);
int         sfsscanf(const char* s, const char* format, ...);
int         sfvscanf(Sfile_t* f, const char* format, va_list args);

int         sfputc(Sfile_t* f, int c);
int         sfnputc(Sfile_t* f, int c, int n);
int         sfputu(Sfile_t* f, ulong v);
int         sfputl(Sfile_t* f, long v);
int         sfputd(Sfile_t* f, double v);
int         sfputs(Sfile_t* f, const char* s, int c);
int         sfwrite(Sfile_t* f, const char* buf, int n);
int         sfmove(Sfile_t* fr, Sfile_t* fw, long n, int rsc);
int         sfprintf(Sfile_t* f, const char* format, ...);
int         sfsprintf(char* s, int size, const char* format, ...);
int         sfvprintf(Sfile_t* f, const char* format, va_list args);

void        sfnotify(void (*notify)(Sfile_t* f, int type, int fd));
int         sfsetfd(Sfile_t* f, int fd);
int         sfset(Sfile_t* f, int flags, int i);
```

```
char*       sfsetbuf(Sfile_t* f, char* buf, int size);
int         sffileno(Sfile_t* f);
int         sfeof(Sfile_t* f);
int         sferror(Sfile_t* f);
int         sfclearerr(Sfile_t* f);
int         sfclrlock(Sfile_t* f);

int         sfslen();
int         sfulen(ulong v);
int         sfllen(long v);
int         sfdlen(double v);

long        sfsize(Sfile_t* f);
long        sfseek(Sfile_t* f, long addr, int offset);
long        sftell(Sfile_t* f);

char*       sfecvt(double v, int n, int* decpt, int* sign);
char*       sffcvt(double v, int n, int* decpt, int* sign);
```

DESCRIPTION

Sfio is library of functions to perform I/O on objects called streams. Each stream is of the type `Sfile_t` defined in the header file `<sfio.h>` and corresponds to either a file descriptor (see *open(2)*) or a piece of memory. Stream complexes can be built by stacking streams on one another. Automatic stream synchronization when I/O operations are switched among related streams can be done via stream pooling. On a per stream basis, system call I/O primitives can be changed by stacking I/O disciplines.

During an I/O request, a system call or its discipline replacement may be invoked to fill the stream buffer. Such a function is said to cause an exception if its return value is non-positive. Unless an exception handler has been defined (see `sfdisc()`), *Sfio* will examine `errno` (see `errno.h`) for further actions. If `errno` is `EINTR`, the respective function will be resumed. Otherwise, an error condition is returned.

A stream is normally locked during any I/O operation to prevent concurrent access. Any attempt to access a locked stream results in error. *Sfio* functions return either integer or pointer values. In the event of an error, a function that returns integer values will return `-1` while a function that returns a pointer value will return `NULL`.

A number of bit flags define stream types and their operations. Following are the flags:

`SF_READ`: The stream is readable.

`SF_WRITE`: The stream is writable.

`SF_STRING`: The stream is a string (a byte array) that is readable if `SF_READ` is specified or writable if `SF_WRITE` is specified.

`SF_APPEND`: The stream is a file opened for appending data. This means that data written to the stream is always appended at the end of the file. On systems where there is no primitive to specify at file opening time that a file is opened for append only, `lseek()` (or its discipline replacement) will be used on the file stream to approximate this behavior.

`SF_LINE`: The stream is line-oriented. For write streams, this means that the buffer is flushed whenever a new-line character is output. For read streams, this means that `sfpeek()` (below) will return a buffer of data which ends with a new-line. In this case, the amount of data that can be returned is limited by the buffer size.

`SF_MALLOC`: The buffer was obtained via `malloc()` and can be reallocated or freed by *Sfio*.

`SF_SHARE`: The associated stream is a file stream that may be operated on by means beyond

straightforward *Sfio* usage, for example a stream shared by multiple processes. In this case, each read or write system call will be preceded by a `lseek()` (or its discipline replacement) to ensure that the logical stream location corresponds to the physical file location.

`sfnew(f,buf,size,fd,flags)` is the primitive for creating or renewing streams. `f`, if not `NULL`, is an existing stream to be reused. This stream will be closed before renewing but its pool and discipline stack remain unchanged. If `f` is `NULL` or cannot be closed, a new stream is created. `buf` and `size` determines a buffering scheme. See `sfsetbuf()` for more details. `fd` is a file descriptor (e.g., from *open()*) to use for I/O operations if the stream is not an `SF_STRING` stream. `flags` is a bit vector composing from the bit flags described above.

`sfopen(f,string,mode)` is a high-level function based on `sfnew()` to create new streams from files or strings. `f` is treated in the same way as `sfnew()` treats its counterpart. `mode` can be any one of: `"r"`, `"r+"`, `"w"`, `"w+"`, `"a"`, `"a+"`, `"s"` and `"s+"`. The `'r'`, `'w'`, and `'a'` specify read, write and append mode for file streams. In these cases, `string` defines a path name to a file. The `'s'` specifies that `string` is a `0` terminated string to be opened for read. The `'+'` means that the new stream will be opened for both reading and writing.

`sfdopen(fd,mode)` makes a stream using the file descriptor `fd`. `mode` is used in the same way as in `sfopen()`.

`sfpopen(cmd,mode,fcomp)` opens a stream `f` which is a pipe to (from) the command `cmd` if the mode is `"w"` (`"r"`). If the mode is `"w+"` or `"r+"`, another stream for the opposite operation is created and returned in `fcomp`. Note that if either of these streams is closed, the other is also closed.

`sfstack(base,top)` is used to push or pop stream stacks. Each stream stack is identified by a `base` stream via which all I/O operations are performed. Other streams on the stack are locked so that operations that may change their internal states are forbidden. The type of operations that can be done on a stack is defined by the top level stream. If an I/O operation is performed and the top level stream reaches the end of file condition or an error condition other than interrupts, it is automatically popped and closed (see also `sfdisc()` for alternative handling of these conditions). `top`, if not `NULL`, is a stream to be pushed on top of the stack. In this case, the `base` stream pointer is returned. If `top` is `NULL`, the stack is popped and the pointer to the popped stream is returned.

`sfpushed(f)` returns the pointer to the stream pushed below `f`.

`sftmp(size)` creates a stream for writing and reading temporary data. If `size` is negative, the stream is a pure `SF_STRING` stream. Otherwise, the stream is first created as a `SF_STRING` stream with a buffer of length `size`. A discipline is set so that when this buffer is exhausted, a real temporary file is created. The temporary file is also created on any attempt to change this discipline.

`sfdisc(f,disc)` manipulates the discipline stack of the stream `f`. If `disc` is not `NULL`, it is the address of a new discipline to be pushed onto the stack. Note that the memory space of this discipline is used on a per stream basis. An application should take care to allocate different space for different use of a discipline. If `disc` is `NULL`, the top element of the stack, if any, is popped. If `sfdisc()` is successful, it returns the address of the new discipline for the pushing case, or the address of the popped discipline in the other case. `sfdisc()` returns `NULL` on failure.

`disc` is a pointer to a discipline structure of type `Sfdisc_t` which defines alternative functions for read, write, seek, and to handle exceptions. Each file stream starts with the unremovable discipline that consists of the system calls `read()`, `write()`, and `lseek()`. `Sfdisc_t` contains the following public fields:

```
int     (*readf)();
int     (*writef)();
long    (*seekf)();
int     (*exceptf)();
```

The first three fields of `Sfdisc_t` specify the alternative I/O functions. If any of them is `NULL`, it is inherited from a discipline earlier on the stack. The arguments to the I/O discipline functions have the same meaning as that of the functions `sfrd()`, `sfwr()` and `sfsk()` to be described later.

The exception function, `(*exceptf)()`, if provided, is used to process exceptions. It is called as: `(*exceptf)(f,type,disc)`. Here, `disc` is the discipline that defines the exception and `type` is the exception. Currently, `type` can take on the following values:

> `SF_READ`: a read operation returns a zero or negative value.
>
> `SF_WRITE`: a write operation returns a zero or negative value.
>
> `SF_SEEK`: a seek operation returns a negative value.
>
> `SF_CLOSE`: the stream is being closed.
>
> `SF_DPUSH`: the discipline is about to be pushed down by a new discipline.
>
> `SF_DPOP`: the discipline is about to be popped from the discipline stack.

The return value of `(*exceptf)()` is examined for further actions. A negative value causes the calling function to return with an appropriate error value. A positive value indicates that the exception is to be ignored and the calling function will repeat the respective I/O operation. A zero value causes the calling function to execute certain default actions with respect to the exception. For example, if the stream is a stacked stream, and the exception is a `SF_READ` or `SF_WRITE`, the stream stack is popped and the popped stream is closed. Note that a `SF_STRING` stream does not perform external I/O so the I/O discipline functions are never used. However, an exception occurs whenever an I/O operation exceeds the stream buffer boundary and `(*exceptf)()`, if defined, will be called as appropriate.

`sfrd(f,buf,n,disc)`, `sfwr(f,buf,n,disc)`, and `sfsk(f,addr,offset,disc)` are functions to be used within a discipline function to perform I/O. The discipline stack can be viewed as a set of filters to process incoming data. These functions provides a safe method to use the processing power of earlier discipline functions and to properly handle exceptions. `sfrd()` and `sfwr()` return the number of bytes read or written. `sfsk()` returns the new seek location. On errors, all three functions return a negative value (`-1` or the value returned by the exception handler).

`sfclose(f)` closes the given stream `f` and frees up its resources. If `f` is `NULL`, all streams are closed. If `f` is a stack of streams, all streams on the stack are closed. If `f` is a `sfpopen` stream, its companion stream, if any, is also closed. In this case, `sfclose()` will wait until the associated command terminates, then return its exit status. When a stream is closed, its corresponding file, if any, will be synchronized (see `sfsync()`). In addition, if the stream has a non-trivial discipline stack, the exception functions of the disciplines will be processed twice. In the first pass, the exception functions are called in the bottom to top order with `type` being `SF_CLOSE` (see `sfdisc()`). In the second pass, they are called from top down with `type` being `SF_DPOP`. Space allocated for disciplines can be released in this phase.

`sfsync(f)` synchronizes the physical file pointer of `f` and the logical position in the stream. For a write stream, this means to write out any buffered data. For a read stream, if the stream is seekable, the physical file pointer is moved back if necessary. If `f` is the base of a stack of streams, all streams on the stack are synchronized. Further, a stacked stream can only be synchronized via its base stream.

`sfpool(f,poolf,mode)` manages pools of streams. In a pool of streams, only one stream is current. A stream becomes current when it is used for some I/O operation. When a new stream is to become current, the current stream is synchronized (see `sfsync()`) if its type matches the type of the pool. `f` is the stream to insert or delete from a pool. `poolf` determines the operation to be done on `f`. If `poolf` is `NULL`, `f` is deleted from its current pool. Otherwise, `f` is put into the same pool with `poolf`. If `poolf` is already in a pool, `mode` is ignored. Otherwise, `mode` determines the type of the new pool. It can be constructed from the bits `SF_READ` and `SF_WRITE`.

sfpeek(f,bufp) provides a safe method for accessing the internal buffer of a stream. If bufp is NULL, only the size of the remainder of the buffer is returned. The stream remains untouched. If bufp is not NULL, *bufp will point to the part of the buffer available for a future read or write. If there is no currently available portion in the buffer, the stream will be flushed or filled as appropriate. The return value of sfpeek() indicates how much data or space is available. However, if the stream is in SF_LINE|SF_READ mode, the return value will be the data length up to and including the new-line character. Note that the buffer location is not advanced by sfpeek(). That must be done by a regular I/O call such as sfread or sfwrite on the pointer returned in bufp. sfpeek() treats a read/write stream like a read stream unless the SF_WRITE attribute has been turned off (see sfset()). Generally, the buffer returned by sfpeek() for a read stream should not be assigned values unless the stream was originally opened for both read and write. For file streams on a system with memory mapping, if the stream was opened for both read and write, the buffer may be a shared memory opened in PROT_SHARED mode (see *mmap*(2)). In this case, the application should take care with changing the content of the buffer if it does not wish such changes to be visible in the file system.

sfgetc(f) returns a byte from the stream f or -1 on end-of-file or error.

sfungetc(f,c) pushes c back into the stream f and makes it available on the next read. For efficiency, if c matches the last read byte in the buffer, the buffer pointer is simply backed up (note the effect on sftell() and sfseek(). There is no theoretical limit on the number of bytes that can be pushed back into a stream. Pushed back bytes that were not part of the buffer as noted will be discarded on any operation that require buffer synchronization, e.g., sftell(), sfseek(), and sfsync().

sfgetu(f), sfgetl(f), and sfgetd(f) return an *unsigned long*, *long* , or *double* value that was coded by sfputu(), sfputl(), and sfputd(). If there is not enough data to decode a value, these functions will return -1 and the stream is set in an error state (see sferror()).

sfgets(f,buf,size) reads a line of input from the stream f. If buf is not NULL and size is positive, sfgets reads up to size-1 characters into the buffer buf. Otherwise, the characters are read into a static area that is dynamically grown as necessary. Thus, in this case, a line can have arbitrary length. A 0 character is appended after the input characters. sfgets() returns the pointer to the new string or NULL when no data was read due to end of file or an error condition. After a string is read, its length can be found using sfslen().

sfread(f,buf,n) reads up to n bytes from the stream f and stores them in the given buffer buf. It returns the number of bytes actually read or -1 on error.

sfscanf(f,format,...) scans a number of items from the stream f. The item types are determined from the string format. See *fscanf()* (UNIX User's Manual, Section 3) for details on predefined formats. The standardly supported formats are: i, I, d, D, u, U, o, O, x, X, f, F, e, E, g, G, c, %, s, and []. For security, the formats C, lc, S, ls, {} and l[] are supported. These are the same as c, s, and [] but in the argument list to sfscanf(), a buffer and its size must be defined in a pair. Note that specifying a buffer size does not affect the scan length. If the buffer size is n, only the first n-1 byte of the scanned data will be copied, the rest is discarded. sfscanf() supports three extension formats: %&, %@, and %:.

> The pattern %& indicates that the next argument in the argument list of sfscanf() is a function, say (*extf)(), to process patterns that are not predefined by the sfscanf() interface. The prototype of (*extf)() is:
>
> ```
> int (*extf)(Sfile_t* f, int fmt, int length, char** rv);
> ```
>
> f is the same input stream passed to sfvscanf. fmt is the pattern to be processed. length, if non-negative, is the maximum number of input bytes to be read in processing the pattern, rv is used to return the address of the value to be assigned. (*extf)() returns the size of the value to be assigned. A negative return value from (*extf)() means that the specified pattern cannot be handled. This pattern is treated as if it is not matched.
>
> The pattern %@ indicates that the next argument in the argument list args is a function, say (*argf)(), to process the values of matched patterns. The prototype of (*argf)() is:

```
int (*argf)(int fmt, char* value, int n);
```

If the return value of `(*argf)()` is negative, the processing of the current format string will be stopped (see `%$` below). `fmt` determines the type of `value`: `f` for *float*, `F` for *double*, `h` for *short*, `d` for *int*, `D` for *long*, `s` for *char**. Any other value for `fmt` means that it is an extended pattern and `value` contains an address to the scanned value. `n` contains the size of the object if it is a primitive type. If the object is `char*` or the address of the scanned value of an extended format, `n` is the length of this object.

The pattern `%:` indicates that the next two arguments in the argument list `args` define a new pair of format string and a list of arguments of the type `va_list` (see `varargs.h` or `stdarg.h`). The new pair is pushed on top of the stack and the scanning process continues with them. The top pair of format string and argument list is popped when the processing of the format string is stopped. When a new pair is stacked, `(*argf)()` and `(*extf)()` are inherited. They are reset when the stack is popped.

`sfsscanf(s,format,...)` is similar to `sfscanf()` but it scans data from the string `s`.

`sfvscanf(f,format,args)` is the primitive underlying `sfscanf()` and `sfscanf()` and provides a portable variable argument interface. Programs that use `sfvscanf()` must include either of `varargs.h` or `stdargs.h` as appropriate.

`sfputc(f,c)` writes the byte `c` to the stream `f`.

`sfnputc(f,c,n)` writes `c` to `f` `n` times. It returns the number of bytes written.

`sfputu(f,v)`, `sfputl(f,v)` write the *unsigned long* or *long* value `v` in a format that is byte-order transparent. `sfputd(f,v)` writes the *double* value `v` in a portable format. Portability across two different machines requires that the bit order in a byte is the same on both machines and the size of the primitive type on the output machine is less than or equal to that on the input machine. `sfputd()` relies on the functions `ldexp()` and `frexp()` (See *frexp.3*) for coding. Upon success, `sfputu()`, `sfputl()` and `sfputd()` return the number of bytes output.

`sfputs(f,s,c)` writes the `0` terminated string `s` to the stream `f`. If `c` is non-negative, it is a character to be appended after the string has been output. `sfputs()` returns the number of bytes written.

`sfwrite(f,buf,n)` writes out `n` bytes from the buffer `buf` to the stream `f`. It returns the number of bytes written.

`sfmove(fr,fw,n,rsc)` moves `n` objects from the stream `fr` to the stream `fw`. If either `fr` or `fw` is `NULL`, it acts as if it is a stream corresponding to */dev/null*. If `n` is `<0`, all of `fr` is moved. If `rsc` is non-negative, it defines a record separator. In this case, the objects to be moved are records separated by `rsc`. If `rsc` is negative, the objects are bytes. `sfmove()` returns the number of objects moved.

`sfprintf(f,format,...)` writes out data in a format as defined by the string `format`. See `fprintf()` for details on the predefined formats: `n, s, c, %, h, i, d, p, u, o, x, X, g, G, e, E, f,` and `F`. `sfprintf()` supports additional formats as described below.

The pattern `%&` indicates that the next argument is a function, say `(*extf)()`, to interpret patterns not yet defined by `sfprintf()`. The prototype of `(*extf)()` is:

```
int (*extf)(void* value, int fmt, int precis, char** sp);
```

`value` is the value to be formatted. `fmt` is the pattern to format the value. `precis` is the amount of precision required. `sp` is used to return the address of a string containing the formatted value. If upon returning from `(*extf)()`, `*sp` is `NULL`, the pattern `fmt` is treated as if it is not matched. Otherwise, the return value of `(*extf)()`, if nonnegative, is taken as the length of the string returned in `sp`. If not, the string is considered `0` terminated. The string `*sp` is processed as if the pattern `'s'` was specified.

The pattern `%@` indicates that the next argument is a function, say `(*argf)()`, to get arguments. As long as `(*argf)()` is defined, the argument list is ignored. The prototype of

(*argf)() is:

```
int (*argf)(int fmt, char* val);
```

fmt is the pattern to be processed. Following are ASCII characters and corresponding types: @ for getting a new (*argf)(), & for getting a new (*extf)(), 1 for getting a new format string for stacking, 2 for getting a new argument list for stacking, d for *int*, D for *long*, f for *float*, F for double, and s for *char**. If (*extf)() is defined, and an undefined pattern is encountered, (*argf)() will be called with this pattern. val is an address to store the value to be formatted. The return value of (*argf)(), if negative, stops the processing of the current format (see below).

The pattern %: indicates that the next two arguments define a pair of format string and argument list of the type va_list. If the argument getting function (*argf)() is already defined, it is called with the argument fmt being the characters 1 and 2 for the new format string and argument list respectively. The new pair is stacked on top and processing continue from there. The top pair of format string and argument is popped when the format string is exhausted. When a new pair is pushed, (*argf)() and (*extf)() are inherited. When a pair is popped, these functions will be reset.

sfsprintf(s,size,format,...) is similar to sfprintf() but it is used to format the character array s which is of size size. The length of the resulting string can be obtained with sfslen().

sfvprintf(f,format,args) is the primitive underlying sfprintf() and sfsprintf(). It provides a portable variable argument interface. Programs that use sfvprintf() must include either of varargs.h or stdargs.h as appropriate.

sfnotify(notify) sets a function (*notify)() which will be called whenever a stream is created, closed or when its file descriptor is changing (see sfsetfd()). (*notify)() is called with three arguments. The first argument is the stream pointer and the second argument is one of SF_NEW, SF_CLOSE or SF_SETFD to indicate whether the stream is being opened, or closed or if its file descriptor is changing. The third argument is the new file descriptor if type is SF_SETFD, otherwise, it is the current file descriptor.

sfsetfd(f,fd) changes the file descriptor of the file stream f. If the current file descriptor and fd are both non-negative, f's descriptor will be changed to a value larger or equal to fd. If the change is successful, the previous file descriptor will be closed. If fd is negative, the stream is synchronized (see sfsync()) and its file descriptor will be set to this value. The stream will remain inaccessible until a future sfsetfd() call to reset the file descriptor to a non-negative value. If the current file number is negative and fd is non-negative, the stream will be reinitialized. (*notify)(f,SF_SETFD,fd) is called immediately before the file descriptor is changed (see sfnotify()). sfsetfd() returns fd on success and -1 otherwise.

sfset(f,flags,i) sets flags for the stream f. flags defines a collection of flags to be turned on or off depending on whether i is non-zero or zero. The flags that can be set are: SF_READ, SF_WRITE, SF_LINE, SF_MALLOC and SF_SHARE. Note that SF_READ and SF_WRITE can be set only if the stream was opened as SF_READ|SF_WRITE. Turning off one of them means that the stream is to be treated as if it was opened with the other flag exclusively (see sfpeek()). It is not possible to turn off both. sfset() returns the entire set of flags controlling the stream. Thus, the current set of flags can be found by sfset(f,0,0).

sfsetbuf(f,buf,size) changes the current buffer of the stream f to the new buffer buf. The stream will be synchronized before the buffer is changed. If size is positive, buf is taken as a buffer of the given size. If size is zero, the stream will be unbuffered. If size is negative, buf is not used and some internal buffering scheme is used, e.g., memory mapping. If a new buffer is successfully set and the old buffer has not been deallocated, sfsetbuf() returns the address of the old buffer. Otherwise, it returns NULL.

`sffileno(f)` returns the file descriptor of the stream `f`.

`sfeof(f)` tells whether there is any more data in the stream `f`.

`sfsize(f)` returns the size of the stream `f` (see `sfnew()`). If the stream is not seekable or if the size is not determinable, `sfsize()` returns `-1`.

`sferror(f)` and `sfclearerr(f)` returns and clears the error condition on `f`. Note that the error condition of a stream is only informative. It does not prevent I/O operations from being performed.

`sfclrlock(f)` clears the lock on a locked stream. Though this is unsafe, it is useful for emergency access or to clear a stream left locked because of non-local jumps (e.g., `longjmp()`).

`sfslen()` returns the length of the string most recently obtained via a `sfgets()`, `sfsprintf()`, `sfecvt()` or `sffcvt()` call.

`sfulen(v)`, `sfllen(v)` and `sfdlen(v)` return the number of bytes required to code the *unsigned long*, *long* or *double* value `v`.

`sfseek(f,addr,offset)` sets the next read/write location for the stream `f` at a new address defined by `addr` and `offset`. If `offset` is `0`, `addr` is the desired address. If `offset` is `1`, `addr` is offset from the current location. Note that if `f` was opened for appending (`SF_APPEND`) and the last operation done on it was a write operation, the *current location* is at the physical end of file. If `offset` is `2`, `addr` is offset from the *physical* end of the stream. See also `sfungetc()`.

`sftell(f)` returns the current location in the stream `f`. As with `sfseek()`, if `f` was opened for appending (`SF_APPEND`) and the last operation done on it was a write operation, the current location is at the physical end of file. If the stream `f` is unseekable, `sftell` returns the number of bytes read from or written to `f`. See also `sfungetc()`

`sfecvt(v,n,decpt,sign)` and `sffcvt(v,n,decpt,sign)` are functions to convert floating values to ASCII. They corresponds to the standard functions `ecvt()` and `fcvt()`. The length of the conversion string most recently done by `sfecvt()` or `sffcvt()` can be found by `sfslen()`.

AUTHORS

Phong Vo (att!ulysses!kpv) and David G. Korn (att!ulysses!dgk).

David Korn received a B.S. in Mathematics in 1965 from Rensselaer Polytechnic Institute and a Ph.D. in Mathematics from the Courant Institute at New York University in 1969 where he worked as a research scientist in the field of transonic aerodynamics until joining Bell Laboratories in September 1976. He was a visiting Professor of computer science at New York University for the 1980-81 academic year and worked on the ULTRA-computer project (a project to design a massively parallel super-computer).

He is currently a supervisor of research at Murray Hill, New Jersey. His primary assignment is to explore new directions in software development techniques that improve programming productivity. His best know effort in this area is the Korn shell, ksh, which is a Bourne compatible UNIX shell with many features added. The language is described in a book which he co-authored with Morris Bolsky. In 1987, he received a Bell Labs fellow award.

Phong Vo received an M.A. in Mathematics in 1977 and a Ph.D. in Mathematics in 1981, both from the University of California at San Diego. He joined Bell Labs at Murray Hill, New Jersey in 1981 where he is currently a Distinguished Member of Technical Staff. His research interests include graph theory, data structures and algorithms, and the applications of such theoretical techniques in user interface and software tools. He authored or coauthored a number of popular software tools including a 2-D Gel analysis system used at the NIH and many university research facilities for protein analyses, the System V Release 3.1 <curses> library for screen management, the System V Release 4 malloc package for memory allocation, IFS, a language for building applications with menu and form interfaces, and DAG, a program to draw directed graphs.

8½, the Plan 9 Window System

Rob Pike

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
`rob@research.att.com`

ABSTRACT

The Plan 9 window system, 8½, is a modest-sized program of novel design. It provides ASCII I/O and bitmap graphic services to both local and remote client programs by offering a multiplexed file service to those clients. It serves traditional UNIX files like `/dev/tty` as well as more unusual ones that provide access to the mouse and the raw screen. Bitmap graphics operations are provided by serving a file called `/dev/bitblt` that interprets client messages to perform raster operations. The file service that 8½ offers its clients is identical to that it uses for its own implementation, so it is fundamentally no more than a multiplexer. This architecture has some rewarding symmetries and can be implemented compactly; indeed 8½ is considerably *smaller* than most of its clients.

Introduction

In 1989 I constructed a toy window system from only a few hundred lines of source code using a custom language and an unusual architecture involving concurrent processes [Pike89]. Although that system was rudimentary at best, it demonstrated that window systems are not inherently complicated. Last year, for the new Plan 9 distributed system [Pike90], I applied some of the lessons from that toy project to write, in C, a production-quality window system called 8½. 8½ provides, on black-and-white or grey-scale but not color displays, the services required of a modern window system, including programmability and support for remote graphics. The entire system, including the default program that runs in the window — the equivalent of `xterm` [Far89] with 'cutting and pasting' between windows — is well under 60 kilobytes of text on a Motorola 68020 processor, about half the size of the operating system kernel that supports it and a tenth the size of the X server [Sche86] *without* `xterm`.

What makes 8½ so compact? Much of the saving comes from overall simplicity: 8½ has little graphical fanciness, a concise programming interface, and a simple, fixed user interface. 8½ also makes some decisions by fiat — three-button mouse, overlapping windows, built-in terminal program and window manager, etc. — rather than trying to appeal to all tastes. Although compact, 8½ is not ascetic. It provides the fundamentals and enough extras to make them comfortable to use. The most important contributor to its small size, though, is its overall design as a file server. This structure may be applicable to window systems on traditional UNIX-like operating systems.

The small size of 8½ does not reflect reduced functionality: 8½ provides service roughly equivalent to X windows, other than the lack of support for color terminals. (The provision of that support would not dramatically increase the code size.) 8½'s clients may of course be as complex as they choose, although the tendency to mimic 8½'s design and the clean programming interface means they are not nearly as bloated as X applications.

User's Model

8½ turns the single screen, mouse, and keyboard of the terminal (in Plan 9 terminology) or workstation (in commercial terminology) into an array of independent virtual terminals that may be ASCII terminals supporting a shell and the usual suite of tools or graphical applications using the full power of the bitmap screen and mouse. The entire programming interface is provided through reading and writing files in `/dev`.

Primarily for reasons of history and familiarity, the general model and appearance of 8½ are similar to those of `mux` [Pike88]. The right button has a short menu for controlling window creation, destruction, and placement. When a window is created, it runs the default shell, `rc` [Duff90], with standard input and output directed to the window and accessible through the file `/dev/cons` ('console'), analogous to the `/dev/tty` of UNIX.® The name change represents a break with the past: Plan 9 does not provide a Teletype-style model of terminals. 8½ provides the only way most users ever access Plan 9.

Graphical applications, like ordinary programs, may be run by typing their names to the shell running in a window. This runs the application in the same window; to run the application in a new window one may use an external program, `window`, described below. For graphical applications, the virtual terminal model is extended somewhat to allow programs to perform graphical operations, access the mouse, and perform related functions by reading and writing files with suggestive names such as `/dev/mouse` and `/dev/window` multiplexed per-window much like `/dev/cons`. The implementation and semantics of these files, described below, is central to the structure of 8½.

The default program that runs in a window is familiar to users of Blit terminals [Pike83]. It is very similar to that of `mux` [Pike88], providing mouse-based editing of input and output text, the ability to scroll back to see earlier output, and so on. It also has a new feature, toggled by typing ESC, that enables the user to control when typed characters may be read by the shell or application, instead of (for example) after each newline. This feature makes the window program directly useful for many text-editing tasks such as composing mail messages before sending them.

Plan 9 and 8½

Plan 9 is a distributed system that provides support for UNIX-like applications in an environment built from distinct CPU servers, file servers, and terminals connected by a variety of networks [Pike90]. The terminals are comparable to modest workstations that, once connected to a file server over a medium-bandwidth network such as Ethernet, are self-sufficient computers running a full operating system. Unlike workstations, however, their role is just to provide an affordable multiplexed user interface to the rest of the system: they run the window system and support simple interactive tasks such as text editing. Thus they lie somewhere between workstations and X terminals in design, cost, performance, and function. (The terminals can be used for general computing, but in practice Plan 9 users do their computing on the CPU servers.) The Plan 9 terminal software, including 8½, was developed on a 68020-based machine called a Gnot and has been ported to the NeXTstation, the MIPS Magnum 3000, and the Sun SPARCstation SLC: all small workstations that we use as terminals.

Heavy computations such as compilation, text processing, or scientific calculation are done on the CPU servers, which are connected to the file servers by high-bandwidth networks, typically point-to-point DMA links. For interactive work, these computations can access the terminal that instantiated them. The terminal and CPU server being used by a particular user are connected to the same file server, although over different networks; Plan 9 provides a view of the file server that is independent of location in the network.

The components of Plan 9 are connected by a common protocol based on the sharing of files. All resources in the network are implemented as file servers; programs that wish to access them connect to them over the network and communicate using ordinary file operations. An unusual aspect of Plan 9 is that the *name space* of a process, the set of files that can be accessed by name (for example by an `open` system call) is not global to all processes on a machine; distinct processes may have distinct name spaces. The system provides methods by which processes may change their name spaces, such as the ability to *mount* a service upon an existing directory, making the files of the service visible in the directory. (This is a different operation from its UNIX namesake.) Multiple services may be mounted upon the same

directory, allowing the files from multiple services to be accessed in the same directory. Options to the mount system call control the order of searching for files in such a *union directory.*

The most obvious example of a network resource is a file server, where permanent files reside. There are a number of unusual services, however, whose design in a different environment would likely not be file-based. Many are described elsewhere [Pike90]; some examples are the representation of processes for debugging, much like Killian's process files for the 8th edition [Kill84], and the implementation of the name/value pairs of the UNIX exec environment as files. User processes may also implement a file service and make it available to clients in the network, much like the 'mounted streams' in the 9th Edition [Pres90]. A typical example is a program that interprets an externally-defined file system such as that on a CD-ROM or a standard UNIX system and makes the contents available to Plan 9 programs. This design is used by all distributed applications in Plan 9, including 8½.

8½ serves a set of files in the conventional directory /dev with names like cons, mouse, and screen. Clients of 8½ communicate with the window system by reading and writing these files. For example, a client program, such as a shell, can print text by writing its standard output, which is automatically connected to /dev/cons, or it may open and write that file explicitly. Unlike files served by a traditional file server, however, the instance of /dev/cons served in each window by 8½ is a distinct file; the per-process name spaces of Plan 9 allow 8½ to provide a unique /dev/cons to each client. This mechanism is best illustrated by the creation of a new 8½ client.

When 8½ starts, it creates a full-duplex pipe to be the communication medium for the messages that implement the file service it will provide. One end will be shared by all the clients; the other end is held by 8½ to accept requests for I/O. When a user makes a new window using the mouse, 8½ allocates the window data structures and forks a child process. The child's name space, initially shared with the parent, is then duplicated so that changes the child makes to its name space will not affect the parent. The child then attaches its end of the communication pipe, cfd, to the directory /dev by doing a mount system call:

```
mount(cfd, "/dev", MBEFORE, buf)
```

This call attaches the service associated with the file descriptor cfd — the client end of the pipe — to the beginning of /dev so that the files in the new service take priority over existing files in the directory. This makes the new files cons, mouse, and so on, available in /dev in a way that hides any files with the same names already in place. The argument buf is a character string (null in this case), described below.

The client process then closes file descriptors 0, 1, and 2 and opens /dev/cons repeatedly to connect the standard input, output, and error files to the window's /dev/cons. It then does an exec system call to begin executing the shell in the window. This entire sequence, complete with error handling, is 28 lines of C.

The view of these events from 8½'s end of the pipe is a sequence of file protocol messages from the new client generated by the intervening operating system in response to the mount and open system calls executed by the client. The message generated by the mount informs 8½ that a new client has attached to the file service it provides; 8½'s response is a unique identifier kept by the operating system and passed in all messages generated by I/O on the files derived from that mount. This identifier is used by 8½ to distinguish the various clients so each sees a unique /dev/cons; most servers do not need to make this distinction.

A process unrelated to 8½ may create windows by a variant of this mechanism. When 8½ begins, it uses a Plan 9 service to 'post' the client end of the communication pipe in a public place. A process may open that pipe and mount it to attach to the window system, much in the way an X client may connect to a UNIX domain socket to the server bound to the file system. The final argument to mount is passed through uninterpreted by the operating system. It provides a way for the client and server to exchange information at the time of the mount. 8½ interprets it as the dimensions of the window to be created for the new client. (In the case above, the window has been created by the time the mount occurs, and buf carries no information.) When the mount returns, the process can open the files of the new window and begin I/O to use it.

Because 8½'s interface is based on files, standard system utilities can be used to control its services. For example, its method of creating windows externally is packaged in a 12-line shell script, called `window`, the core of which is just a `mount` operation that prefixes 8½'s directory to `/dev` and runs a command passed on the argument line:

```
mount -b $'8.5serv' /dev
$* < /dev/cons > /dev/cons >[2] /dev/cons &
```

The `window` program is typically employed by users to create their initial working environment when they boot the system, although it has more general possibilities.

Other basic features of the system fall out naturally from the file-based model. When the user deletes a window, 8½ sends the equivalent of a UNIX signal to the process group — the clients — in the window, removes the window from the screen, and poisons the incoming connections to the files that drive it. If a client ignores the signal and continues to write to the window, it will get I/O errors. If, on the other hand, all the processes in a window exit spontaneously, they will automatically close all connections to the window. 8½ counts references to the window's files; when none are left, it shuts down the window and removes it from the screen. As a different example, when the user hits the DEL key to generate an interrupt, 8½ writes a message to a special file, provided by Plan 9's process control interface, that interrupts all the processes in the window. In all these examples, the implementation works seamlessly across a network.

There are two valuable side effects of implementing a window system by multiplexing `/dev/cons` and other such files. First, the problem of giving a meaningful interpretation to the file `/dev/cons` (`/dev/tty`) in each window is solved automatically. To provide `/dev/cons` is the fundamental job of the window system, rather than just an awkward burden; other systems must often make special and otherwise irrelevant arrangements for `/dev/tty` to behave as expected in a window. Second, any program that can access the server, including a process on a remote machine, can access the files using standard read and write system calls to communicate with the window system, and standard open and close calls to connect to it. Again, no special arrangements need to be made for remote processes to use all the graphics facilities of 8½.

Graphical input

Of course 8½ offers more than ASCII I/O to its clients. The state of the mouse may be discovered by reading the file `/dev/mouse`, which returns a ten-byte message encoding the state of the buttons and the position of the cursor. If the mouse has not moved since the last read of `/dev/mouse`, or if the window associated with the instance of `/dev/mouse` is not the 'input focus', the read blocks.

The format of the message is:

> `'m'`
> 1 byte of button state
> 4 bytes of x, low byte first
> 4 bytes of y, low byte first

As in all shared data structures in Plan 9, the order of every byte in the message is defined so all clients can execute the same code to unpack the message into a local data structure.

For keyboard input, clients can read `/dev/cons` or, if they need character-at-a-time input, `/dev/rcons` ('raw console'). There is no explicit event mechanism to help clients that need to read from multiple sources. Instead, a small (336 line) external support library can be used. It attaches a process to the various blocking input sources — mouse, keyboard, and perhaps a third user-provided file descriptor — and funnels their input into a single pipe from which may be read (three types of) events in the traditional style. This package is a compromise. As discussed in a previous paper [Pike89] I prefer to free applications from event-based programming. Unfortunately, though, I see no easy way to achieve this in single-threaded C programs, and am unwilling to require all programmers to master concurrent programming. It should be noted, though, that even this compromise results in a small and easily understood interface. An example program that uses it is given near the end of the paper.

Graphical output

The file `/dev/screen` may be read by any client to recover the contents of the entire screen, such as for printing (see Figure 1). Similarly, `/dev/window` holds the contents of the current window. These are read-only files.

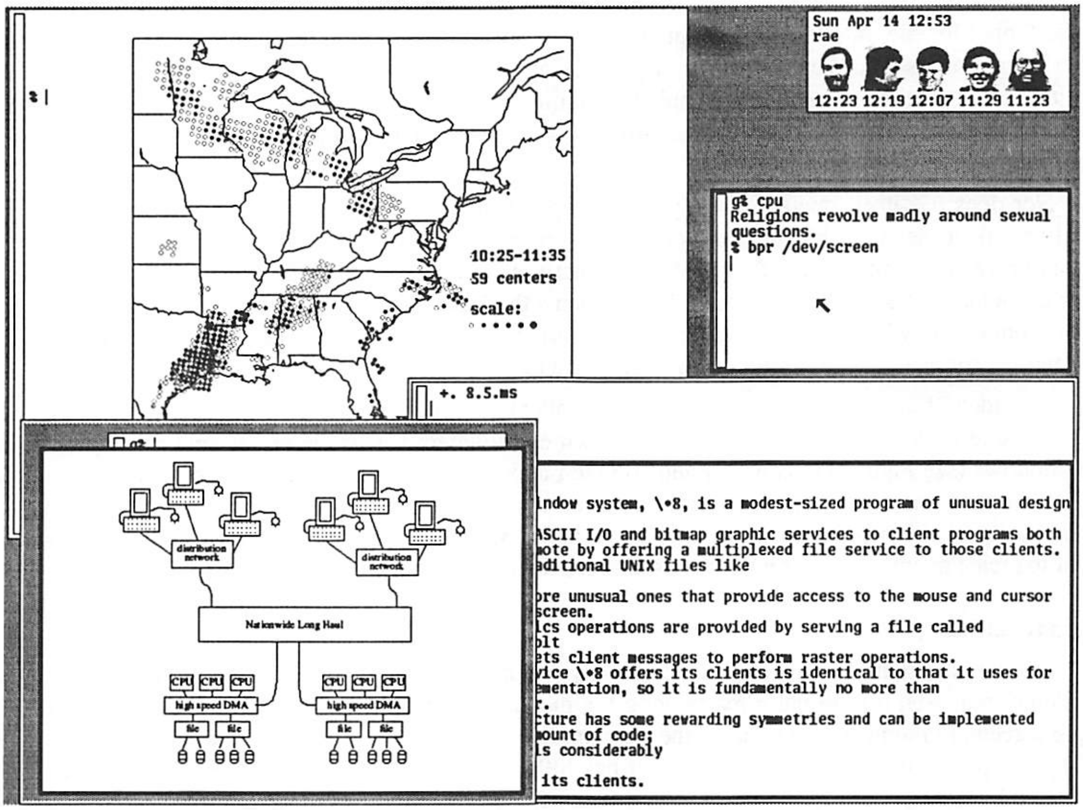


Figure 1. A representative 8½ screen, running on a NeXTstation under Plan 9 (with no NeXT software). In the upper right, a program announces the arrival of mail. In the upper left is a map displayed by the CPU server that shows the radar reflectance of precipitation in the eastern United States. In the lower right there is a screen editor, `sam` [Pike87], and in the lower left an 8½ running recursively and, inside that instantiation, a previewer for `troff` output. Underneath the faces is a small window running the command that prints the screen by passing `/dev/screen` to the bitmap printing utility.

To perform graphics operations in their windows, client programs access `/dev/bitblt`. It implements a protocol that encodes bitmap graphics operations. Most of the messages in the protocol (there are 14 messages in all) are transmissions (via a write) from the client to the window system to perform a graphical operation such as a `bitblt` [PLR85] or character-drawing operation; a few include return information (recovered via a read) to the client. As with `/dev/mouse`, the `/dev/bitblt` protocol is in a defined byte order. Here, for example, is the layout of the `bitblt` message:

> `'b'`
> 2 bytes of destination id
> 2x4 bytes of destination point
> 2 bytes of source id
> 4x4 bytes of source rectangle
> 2 bytes of boolean function code

The message is trivially constructed from the `bitblt` subroutine in the library, defined as

```
void bitblt(Bitmap *dst, Point dp, Bitmap *src, Rectangle sr, Fcode c).
```

The 'id' fields in the message belie another property of 8½: the clients do not store the actual data for any of their bitmaps locally. Instead, the protocol provides a message to allocate a bitmap, to be stored in the server, and returns to the client an integer identifier, much like a UNIX file descriptor, to be used in operations on that bitmap. Bitmap number 0 is conventionally the client's window, analogous to standard input for file I/O. In fact, no bitmap graphics operations are executed in the client at all; they are all performed on its behalf by the server. Again, using the standard remote file operations in Plan 9, this permits remote machines having no graphics capability, such as the CPU server, to run graphics applications. Analogous features of the original Andrew window system [Gos86] and of X [Sche86] require more complex mechanisms.

Nor does 8½ itself operate directly on bitmaps. Instead, it calls another server to do its graphics operations for it, using an identical protocol. The operating system for the Plan 9 terminals contains an internal server that implements that protocol, exactly as does 8½, but for a single client. That server stores the actual bytes for the bitmaps and implements the fundamental bitmap graphics operations. Thus the environment in which 8½ runs has exactly the structure it provides for its clients; 8½ reproduces the environment for its clients, multiplexing the interface to keep the clients separate.

This idea of multiplexing by simulation is applicable to more than window systems, of course, and has some side effects. Since 8½ simulates its own environment for its clients, it may run in one of its own windows (see Figure 1). A useful and common application of this technique is to connect a window to a remote machine, such as a CPU server, and run the window system there so that each subwindow is automatically on the remote machine. It is also a handy way to debug a new version of the window system or to create an environment with, for example, a different default font.

Implementation

To provide graphics to its clients, 8½ mostly just multiplexes and passes through to its own server the clients' requests, occasionally rearranging the messages to maintain the fiction that the clients have unique screens (windows). To manage the overlapping windows it uses the layers model, which is handled by a separate library [Pike83a]. Thus it has little work to do and is a fairly simple program; it is dominated by a couple of modest-sized switch statements to interpret the bitmap and file server protocols. The built-in window program and its associated menus and text-management support are responsible for most of the code.

The operating system's server is also compact; the version for the 68020 processor, excluding the implementation of a half dozen bitmap graphics operations, is 1153 lines of C; the graphics operations are another 1118 lines.

8½ is structured as a set of communicating coroutines, much as discussed in a 1989 paper [Pike89]. One coroutine manages the mouse, another the keyboard, and another is instantiated to manage the state of each window and associated client. When no coroutine wishes to run, 8½ reads the next file I/O request from its clients, which arrive serially on the full-duplex communication pipe. Thus 8½ is entirely synchronous.

The program source is small and compiles in about 10 seconds in our Plan 9 environment. There are ten source files and one `makefile` totaling 3860 lines. This includes the source for the window management process, the cut-and-paste terminal program, the window/file server itself, and a small coroutine library (`proc.c`). It does not include the layer library (another 610 lines) or the library to handle the cutting and pasting of text displayed in a window (898 lines), or the general graphics support library that manages all the non-drawing aspects of graphics — arithmetic on points and rectangles, memory management, error handling, clipping, — plus events and non-primitive drawing operations such as circles and ellipses (a final 1797 lines). Not all the pieces of these libraries are used by 8½ itself; a large part of the graphics library in particular is used only by clients. Thus it is somewhat unfair to 8½ just to sum these numbers, including the 2271 lines of support in the kernel, and arrive at a total implementation size of 9436 lines of source to implement all of 8½ from the lowest levels to the highest. But that number gives a fair measure of the complexity of the overall system.

The implementation is also efficient. 8½'s performance is competitive to X windows'. Compared using Dunwoody's and Linton's gbench benchmarks on the 68020, distributed with the "X Test Suite", circles and arcs are drawn about half as fast in 8½ as in X11 release 4 compiled with gcc for equivalent hardware, probably because they are currently implemented in a user library by calls to the point primitive. Line drawing speed is about equal between the two systems. Text is drawn about 20% faster in 8½ and the bitblt test is about four times faster. These numbers vary enough to caution against drawing sweeping conclusions, but they suggest that 8½'s architecture does not penalize its performance. Finally, 8½ boots in under a second and creates a new window apparently instantaneously.

An example

Here is a complete program that runs under 8½. It prints the string "hello world" wherever the left mouse button is depressed, and exits when the right mouse button is depressed. It also prints the string in the center of its window, and maintains that string when the window is resized.

```
#include <u.h>
#include <libc.h>
#include <libg.h>

void
ereshaped(Rectangle r)
{
        Point p;

        screen.r = r;
        bitblt(&screen, screen.r.min, &screen, r, Zero); /* clear window */
        p.x = screen.r.min.x + Dx(screen.r)/2;
        p.y = screen.r.min.y + Dy(screen.r)/2;
        p = sub(p, div(strsize(font, "hello world"), 2));
        string(&screen, p, font, "hello world", S);
}

main(void)
{
        Mouse m;

        binit(0, 0);      /* initialize graphics library */
        einit(Emouse);    /* initialize event library */
        ereshaped(screen.r);
        for(;;){
                m = emouse();
                if(m.buttons & RIGHTB)
                        break;
                if(m.buttons & LEFTB){
                        string(&screen, m.xy, font, "hello world", S);
                        /* wait for release of button */
                        do; while(emouse().buttons & LEFTB);
                }
        }
}
```

The complete loaded binary is a little over 18K bytes on a 68020. This program should be compared to the similar ones in the excellent paper by Rosenthal [Rose88]. (The current program does more: it also employs the mouse.) The clumsiest part is ereshaped, a function with a known name that is called from the event library whenever the window is reshaped or moved, as is discovered inelegantly but adequately by a special case of a mouse message. (Simple so-called expose events are not events at all in 8½; the layer library takes care of them transparently.) The lesson of this program, in deference to Rosenthal, is that if the window system is cleanly designed a toolkit should be unnecessary for simple tasks.

Status

8½ is in regular daily use by almost all the 60 people in our research center. Some of those people use it to access Plan 9 itself; others use it as a front end to remote UNIX systems, much as one would use an X terminal. It has been fairly stable for almost a year.

Some things about 8½ may change. It would be nice if its capabilities were more easily accessible from the shell. A companion to this paper [Pike91] proposes one way to do this, but that does not include any graphics functionality. Perhaps an ASCII version of the `/dev/bitblt` file is a way to proceed; that would allow, for example, `awk` programs to draw graphs directly. Finally, 8½ offers no 'iconization' of its clients, and probably never will. But it would be helpful for it to provide some quick mechanism for managing a cluttered screen, and some ideas are being considered. By the time this paper appears, 8½ will probably address this issue. The code is unlikely to grow substantially as a result, however, and clients will be unaffected by the changes.

Can this style of window system be built on other operating systems? A major part of the design of 8½ depends on its structure as a file server. In principle this could be done for any system that supports user processes that serve files, such as any system running NFS or AFS [Sun89, Kaza87]. One requirement, however, is 8½'s need to respond to its clients' requests out of order: if one client reads `/dev/cons` in a window with no characters to be read, other clients should be able to perform I/O in their windows, or even the same window. Another constraint is that the 8½ files are like devices, and must not be cached by the client. NFS cannot honor these requirements; AFS may be able to. Of course, other interprocess communication mechanisms such as sockets could be used as a basis for a window system. One may even argue that X's model fits into this overall scheme. It may prove easy and worthwhile to write a small 8½-like system for commercial UNIX systems to demonstrate that its merits can be won in systems other than Plan 9.

Conclusion

In conclusion, 8½ uses an unusual architecture in concert with the file-oriented interprocess communication of Plan 9 to provide network-based interactive graphics to client programs. It demonstrates that even production-quality window systems are not inherently large or complicated and may be simple to use and to program.

Acknowledgements

Helpful comments on early drafts of this paper were made by Doug Blewett, Stu Feldman, Chris Fraser, Brian Kernighan, Dennis Ritchie, and Phil Winterbottom. Many of the ideas leading to 8½ were tried out in earlier, sometimes less successful, programs. I would like to thank those users who suffered through some of my previous 7½ window systems.

References

[Duff90] Tom Duff, "Rc - A Shell for Plan 9 and UNIX systems", Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 21-33

[Far89] Far too many people, XTERM(1), Massachusetts Institute of Technology, 1989

[Gos86] James Gosling and David Rosenthal, "A window manager for bitmapped displays and UNIX", in Methodology of Window Management, edited by F.R.A. Hopgood et al., Springer, 1986

[Kaza87] Mike Kazar, "Synchronization and Caching issues in the Andrew File System", Tech. Rept. CMU-ITC-058, Information Technology Center, Carnegie Mellon University, June, 1987

[Kill84] Tom Killian, "Processes as Files", USENIX Summer Conf. Proc., Salt Lake City June, 1984

[Pike83] Rob Pike, "The Blit: A Multiplexed Graphics Terminal", Bell Labs Tech. J., V63, #8, part 2, pp. 1607-1631

[Pike83a] Rob Pike, "Graphics in Overlapping Bitmap Layers", Trans. on Graph., Vol 2, #2, 135-160, reprinted in Proc. SIGGRAPH '83, pp. 331-356

[Pike87] Rob Pike, "The Text Editor `sam`", Softw. - Prac. and Exp., Nov 1987, Vol 17 #11, pp. 813-845

[Pike88] Rob Pike, "Window Systems Should Be Transparent", Comp. Sys., Summer 1988, Vol 1 #3, pp. 279-296

[Pike89] Rob Pike, ''A Concurrent Window System'', Comp. Sys., Spring 1989, Vol 2 #2, pp. 133-153

[Pike90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, ''Plan 9 from Bell Labs'', Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 1-9

[Pike91] Rob Pike, ''A Minimalist Global User Interface'', USENIX Summer Conf. Proc., Nashville, June, 1991, this volume

[PLR85] Rob Pike, Bart Locanthi and John Reiser, ''Hardware/Software Tradeoffs for Bitmap Graphics on the Blit'', Softw. - Prac. and Exp., Feb 1985, Vol 15 #2, pp. 131-152

[Pres90] David L. Presotto and Dennis M. Ritchie, ''Interprocess Communication in the Ninth Edition Unix System'', Softw. - Prac. and Exp., June 1990, Vol 20 #S1, pp. S1/3-S1/17

[Rose88] David Rosenthal, ''A Simple X11 Client Program -or- How hard can it really be to write ''Hello, World''?'', USENIX Winter Conf. Proc., Dallas, Jan, 1988, pp. 229-242

[Sche86] Robert W. Scheifler and Jim Gettys, ''The X Window System'', ACM Trans. on Graph., Vol 5 #2, pp. 79-109

[Sun89] Sun Microsystems, NFS: Network file system protocol specification, RFC 1094, Network Information Center, SRI International, March, 1989.

The Plan 9 system, including the window system and perhaps the help program, is being made available to universities on an as-is basis in source form. If you are interested, please contact:

Rob Pike
Bell Labs 2C524
Murray Hill NJ 07974
rob@research.att.com

Rob Pike is a Member of Technical Staff at AT&T Bell Laboratories in Murray Hill, New Jersey, where he has been since 1980, the same year he won the Olympic silver medal in Archery. In 1981 he wrote the first bitmap window system for UNIX systems, and has since written nine more. With Bart Locanthi he designed the Blit terminal; with Brian Kernighan he wrote *The Unix Programming Environment*. A shuttle mission nearly launched a gamma-ray telescope he designed. He is a Canadian citizen and has never written a program that uses cursor addressing.

A Minimalist Global User Interface

Rob Pike

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
`rob@research.att.com`

ABSTRACT

`Help` is a combination of editor, window system, shell, and user interface that provides a novel environment for the construction of textual applications such as browsers, debuggers, mailers, and so on. It combines an extremely lean user interface with some automatic heuristics and defaults to achieve significant effects with minimal mouse and keyboard activity. The user interface is driven by a file-oriented programming interface that may be controlled from programs or even shell scripts. By taking care of user interface issues in a central utility, `help` simplifies the job of programming applications that make use of a bitmap display and mouse.

Introduction

The problem of designing appropriate graphical user interfaces to the UNIX® system is vexing and largely unsolved, even today, ten years after bitmap displays were first attached to UNIX systems. In the frenzy to exploit the screen and mouse, graphical applications have become major subsystems that sidestep or even subvert some of the properties of UNIX that helped make it popular, in particular its piece-parts, tool-based approach to programming. (Nowadays even the word 'tool' has changed meaning; the 'tool' in 'toolkit' is just a user interface gimmick such as a menu or scroll bar, nothing nearly as advanced as, say, `diff`(1)). Although there have been some encouraging recent attempts, in particular ConMan and Tcl [Haeb88,Oust90], they have taken the form of providing interprocess communication within existing environments, permitting established programs to talk to one another. None has approached the problem structurally. Moreover, they are minor counterexamples to the major trend, which is to differentiate among systems by providing ever larger, fancier, and more monolithic graphics subsystems rather than by increasing the functionality or programmability of the overall system. To the software developer, that trend is disappointing; modern user interface toolkits and window systems are as complex as the systems UNIX displaced with its elegant, simple approach.

`Help` is an experimental program that combines aspects of window systems, shells, and editors to address these issues in the context of textual applications. It is not a 'toolkit'; it is a self-contained program, more like a shell than a library, that joins users and applications. From the perspective of the application, it provides a universal communication mechanism, based on familiar UNIX file operations, that permits small applications — even shell procedures — to exploit the graphical user interface of the system and communicate with each other. For the user, the interface is extremely spare, consisting only of text, scroll bars, one simple kind of window, and a unique function for each mouse button — no widgets, no icons, not even pop-up menus. Despite these limitations, `help` is an effective environment in which to work and, particularly, to program.

`Help's` roots lie in Wirth's and Gutknecht's Oberon system [Wirt89, Reis91]. Oberon is an attempt to extract the salient features of Xerox's Cedar environment and implement them in a system of manageable size. It is based on a module language, also called Oberon, and integrates an operating system, editor, window system, and compiler into a uniform environment. Its user interface is especially simple: by using the mouse to point at text on the display, one indicates what subroutine in the system to execute next. In a normal UNIX shell, one types the name of a file to execute; instead in Oberon one

selects with a particular button of the mouse a module and subroutine within that module, such as `Edit.Open` to open a file for editing. Almost the entire interface follows from this simple idea. The user interface of `help` is in turn an attempt to adapt the user interface of Oberon from its language-oriented structure on a single-process system to a file-oriented multi-process system, Plan 9 [Pike90]. That adaptation must not only remove from the user interface any specifics of the underlying language; it must provide a way to bind the text on the display to commands that can operate on it: Oberon passes a character pointer; `help` needs a more general method because the information must pass between processes. The method chosen uses the standard currency in Plan 9: files and file servers.

The interface seen by the user

This section explains the basics of the user interface; the following section uses this as the background to a major example that illustrates the design and gives a feeling for the system in action.

`Help` operates only on text; at the moment it has no support for graphical output. A three-button mouse and keyboard provide the interface to the system. The fundamental operations are to type text with the keyboard and to control the screen and execute commands with the mouse buttons. Text may be selected with the left and middle mouse buttons. The middle button selects text defining the action to be executed; the left selects the object of that action. The right button controls the placement of windows. Note that typing does not execute commands; newline is just a character.

Several interrelated rules were followed in the design of the interface. These rules are intended to make the system as efficient and comfortable as possible for its *users*. First, *brevity:* there should be no actions in the interface — button clicks or other gestures — that do not directly affect the system. Thus `help` is not a 'click-to-type' system because that click is wasted; there are no pop-up menus because the gesture required to make them appear is wasted; and so on. Second, *no retyping:* it should never be necessary or even desirable to retype text that is already on the screen. Many systems allow the user to copy the text on the screen to the input stream, but for small pieces of text such as file names it often seems easier to retype the text than to use the mouse to pick it up. As a corollary, when not typing genuinely new text, such as when browsing source code or debugging, it should be possible to work efficiently and comfortably without using the keyboard at all. Third, *automation:* let the machine fill in the details and make mundane decisions. For example, it should be good enough just to point at a file name, rather than to pass the mouse over the entire textual string. Finally, *defaults:* the most common use of a feature should be the default. Similarly, the smallest action should do the most useful thing. Complex actions should be required only rarely and when the task is unusually difficult.

The `help` screen is tiled with windows of editable text, arranged in (typically) two side-by-side columns. Figure 1 shows a `help` screen in mid-session. Each window has two subwindows, a single *tag* line across the top and a *body* of text. The tag typically contains the name of the file whose text (or a copy thereof) appears in the body.

The text in each subwindow (tag or body) may be edited using a simple cut-and-paste editor integrated into the system. The left mouse button selects text; the selection is that text between the point where the button is pressed and where it is released. Each subwindow has its own selection. One subwindow — the one with the most recent selection or typed text — is the location of the *current selection* and its selection appears in reverse video. The selection in other subwindows appears in outline.

Typed text replaces the selection in the subwindow under the mouse. The right mouse button is used to rearrange windows. The user points at the tag of a window, presses the right button, drags the window to where it is desired, and releases the button. `Help` then does whatever local rearrangement is necessary to drop the window to its new location (the rule of automation). This may involve covering up some windows or adjusting the position of the moved window or other windows. `Help` attempts to make at least the tag of a window fully visible; if this is impossible, it covers the window completely.

A tower of small black squares, one per window, adorns the left edge of each column. (See Figure 1.) These tabs represent the windows in the column, visible or invisible, in order from top to bottom of the column, and can be clicked with the left mouse button to make the corresponding window fully visible, from the tag to the bottom of the column it is in. A similar row across the top of the columns allows the columns to expand horizontally. These little tabs are an adequate but not especially successful

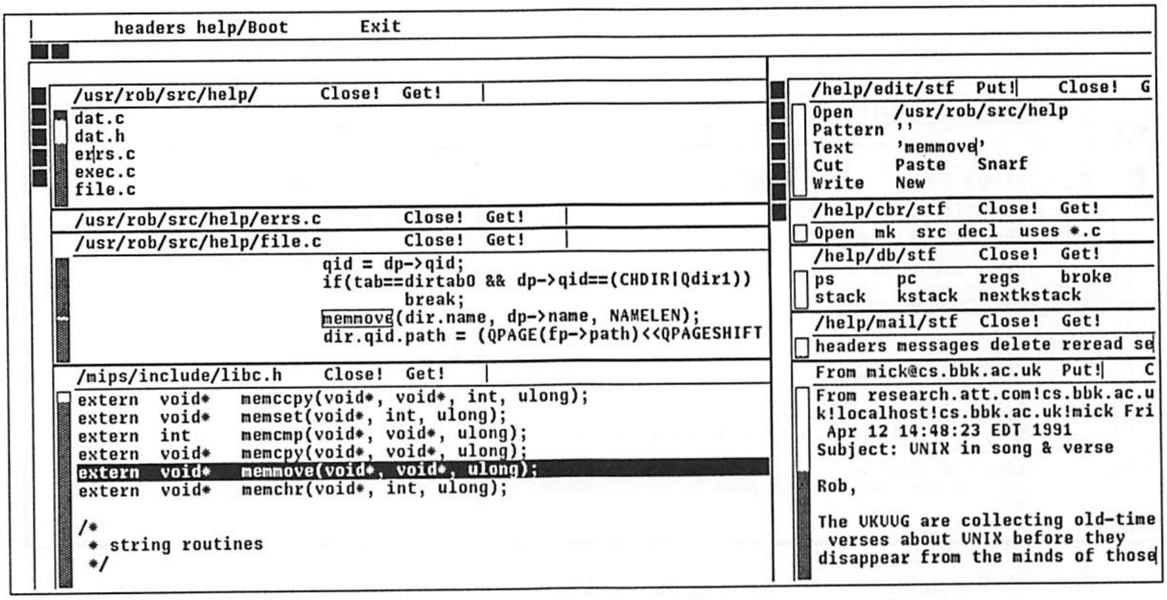


Figure 1: A small `help` screen showing two columns of windows. The current selection is the black line in the bottom left window. The directory `/usr/rob/src/help` has been `Opened` and, from there, the source files `/usr/rob/src/help/errs.c` and `file.c`.

solution to the problem of managing many overlapping windows. The problem needs more work; perhaps the file name of each window should pop up alongside the tabs when the mouse is nearby.

Like the left mouse button, the middle button also selects text, but the act of releasing the button does not leave the text selected; rather it executes the command indicated by that text. For example, to cut some text from the screen, one selects the text with the left button, then selects with the middle button the word `Cut` anywhere it appears on the display. (By convention, capitalized commands represent built-in functions.) As in any cut-and-paste editor, the cut text is remembered in a buffer and may be pasted into the text elsewhere. If the text of the command name is not on the display, one just types it and *then* executes it by selecting with the middle button. Note that `Cut` is not a 'button' in the usual window system sense; it is just a word, wherever it appears, that is bound to some action. To make things easier, `help` interprets a middle mouse button click (not *double* click) anywhere in a word as a selection of the whole word (the rule of defaults). Thus one may just select the text normally, then click on `Cut` with the middle button, involving less mouse activity than with a typical pop-up menu. As a strict rule, if the text for selection or execution is the null string, `help` invokes automatic actions to expand it to a file name or similar context-dependent block of text; if the text is non-null, it is taken literally.

As an extra acceleration, `help` has two commands invoked by chorded mouse buttons. While the left button is still held down after a selection, clicking the middle button executes `Cut`; clicking the right button executes `Paste`, replacing the selected text by the contents of the cut buffer. These are the most common editing commands and it is convenient not to move the mouse to execute them (the rules of brevity and defaults). One may even click the middle and then right buttons, while holding the left down, to execute a cut-and-paste, that is, to remember the text in the cut buffer for later pasting.

More than one word may be selected for execution; executing `Open /usr/rob/lib/profile` creates a new window and puts the contents of the file in it. (If the file is already open, the command just guarantees that its window is visible.) Again, by the rule of automation, the new window's location will be chosen by `help`. The hope is to do something sensible with a minimum of fuss rather than just the right thing with user intervention. This policy was a deliberate and distinct break with most previous systems. (It is present in Oberon and in most tiling window systems but `help` takes it farther.) This is a contentious point, but `help` is an experimental system. The jury is still out but the heuristics continue to be worked on and the need to intervene is diminishing as they improve.

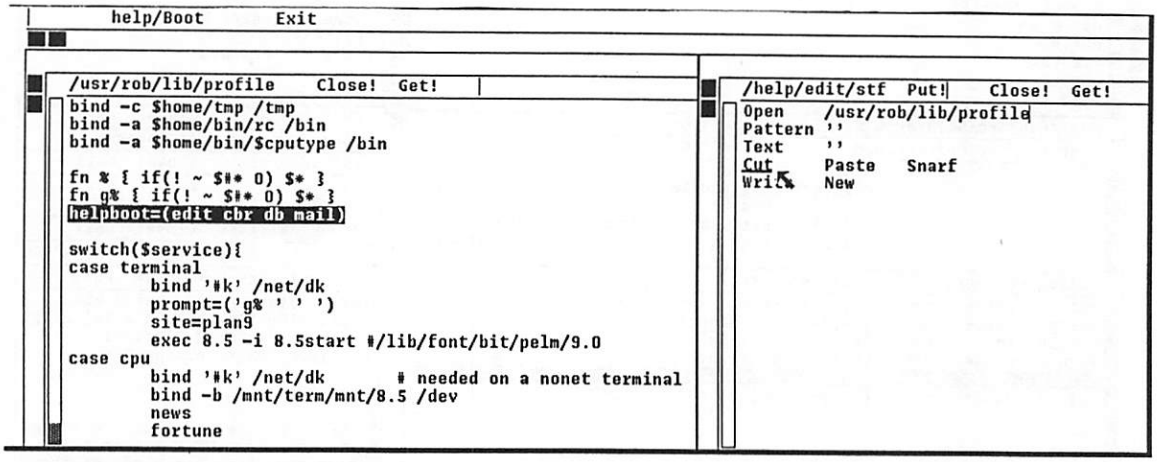


Figure 2: Executing Cut by sweeping the word while holding down the middle mouse button. The text being selected for execution is underlined.

A typical shell window in a traditional window system permits text to be copied from the typescript and presented as input to the shell to achieve some sort of history function: the ability to re-execute a previous command. Help instead tries to predict the future: to get to the screen commands and text that will be useful later. Every piece of text on the screen is a potential command or argument for a command. Many of the basic commands pull text to the screen from the file system with a minimum of fuss. For example, if Open is executed without an argument, it uses the file name containing the most recent selection (the rule of defaults). Thus one may just point with the left button at a file name and then with the middle button at Open to edit a new file. Using all four of the rules above, if Open is applied to a null selection in a file name that does not begin with a slash (/), the directory name is extracted from the file name in the tag of the window and prepended to the selected file name. An elegant use of this is in the handling of directories. When a directory is Opened, help puts the its name, including a final slash, in the tag and just lists the contents in the body. (See Figure 1.)

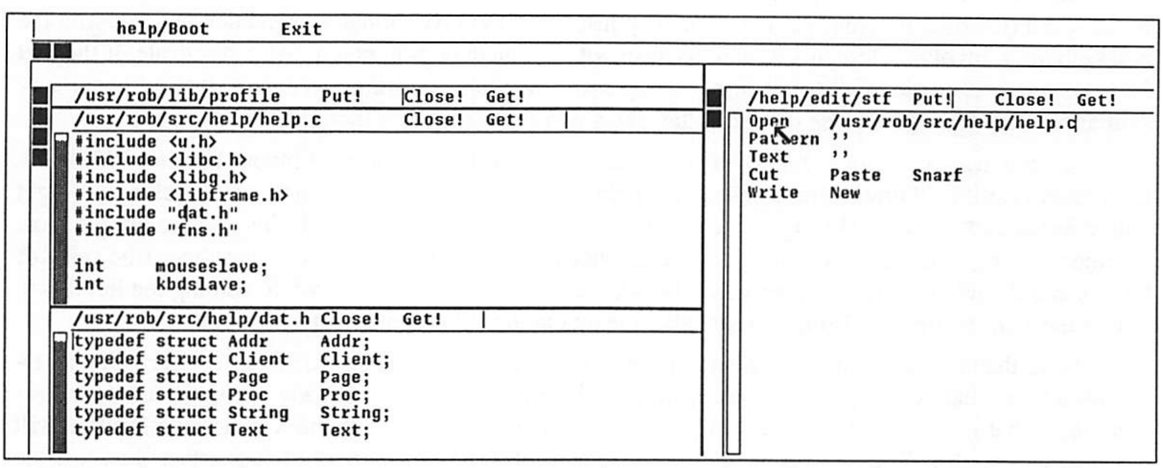


Figure 3: Opening files. After typing the full path name of help.c, the selection is automatically the null string at the end of the file name, so just click Open to open that file; the defaults grab the whole name. Next, after pointing into dat.h, Open will get /usr/rob/src/help/dat.h.

For example, by pointing at `dat.h` in the source file `/usr/rob/src/help/help.c` and executing `Open`, a new window is created containing the contents of `/usr/rob/src/help/dat.h`: two button clicks. (See Figure 3.) Making any non-null selection disables all such automatic actions: the resulting text is then exactly what is selected.

That `Open` prepends the directory name gives each window a context: the directory in which the file resides. The various commands, built-in and external, that operate on files derive the directory in which to execute from the tag line of the window. `Help` has no explicit notion of current working directory; each command operates in the directory appropriate to its operands.

The `Open` command has a further nuance: if the file name is suffixed by a colon and an integer, for example `help.c:27`, the window will be positioned so the indicated line is visible and selected. This feature is reminiscent of Robert Henry's `error`(1) program in Berkeley UNIX, although it is integrated more deeply and uniformly. Also, unlike `error`, `help's` syntax permits specifying general locations, although only line numbers will be used in this paper.

It is possible to execute any external Plan 9 command. If a command is not a built-in like `Open`, it is assumed to be an executable file and the arguments are passed to the command to be executed. For example, if one selects with the middle button the text

```
grep '^main' /sys/src/cmd/help/*.c
```

the traditional command will be executed. Again, some default rules come into play. If the tag line of the window containing the command has a file name and the command does not begin with a slash, the directory of the file will be prepended to the command. If that command cannot be found, the command will be searched for in the conventional directory `/bin`. The standard input of the commands is connected to `/dev/null`; the standard and error outputs are directed to a special window, called `Errors`, that will be created automatically if needed. The `Errors` window is also the destination of any messages printed by the built-in commands.

The interplay and consequences of these rules are easily seen by watching the system in action.

An example

In this example I will go through the process of fixing a bug reported to me in a mail message sent by a user. Please pardon the informal first person for a while; it makes the telling easier.

When `help` starts it loads a set of 'tools', a term borrowed from Oberon, into the right hand column of its initially two-column screen. These are files with names like `/help/edit/stf` (the stuff that the `help` editor provides), `/help/mail/stf`, and so on. Each is a plain text file that lists the names of the commands available as parts of the tool, collected in the appropriate directory. A `help` window on such a file behaves much like a menu, but is really just a window on a plain file. The useful properties stem from the interpretation of the file applied by the rules of `help`; they are not inherent in the file.

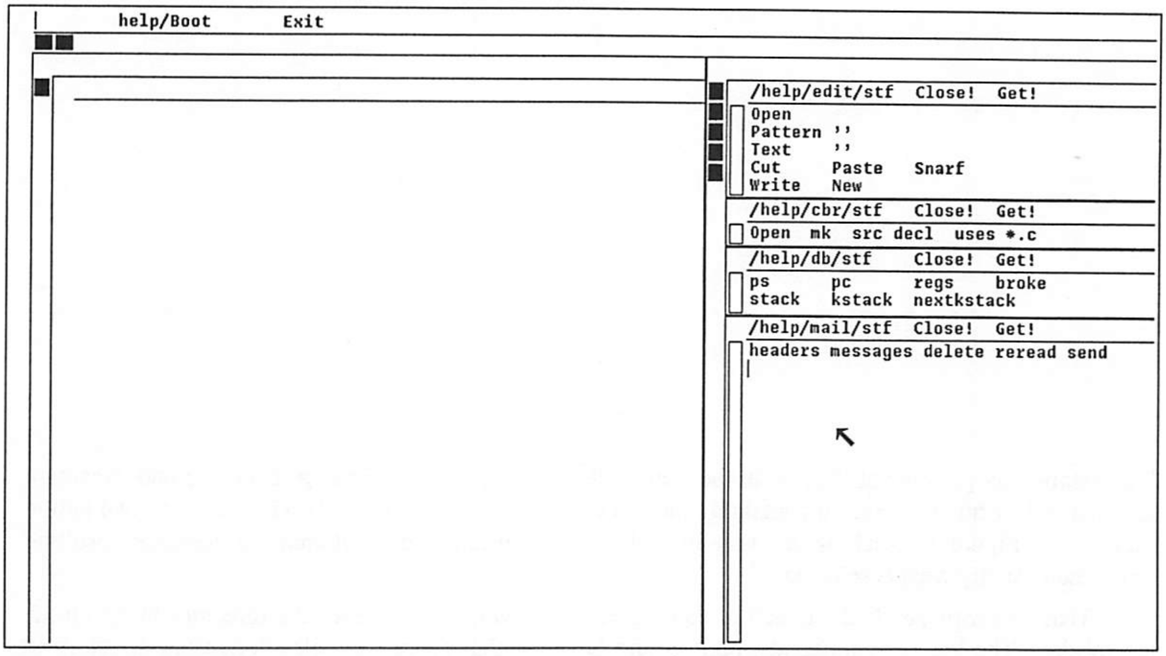


Figure 4: The screen after booting.

To read my mail, I first execute `headers` in the mail tool, that is, I click the middle mouse button on the word `headers` in the window containing the file `/help/mail/stf`. This executes the program `/help/mail/headers` by prefixing the directory name of the file `/help/mail/stf`, collected from the tag, to the executed word, `headers`. This simple mechanism makes it easy to manage a collection of programs in a directory.

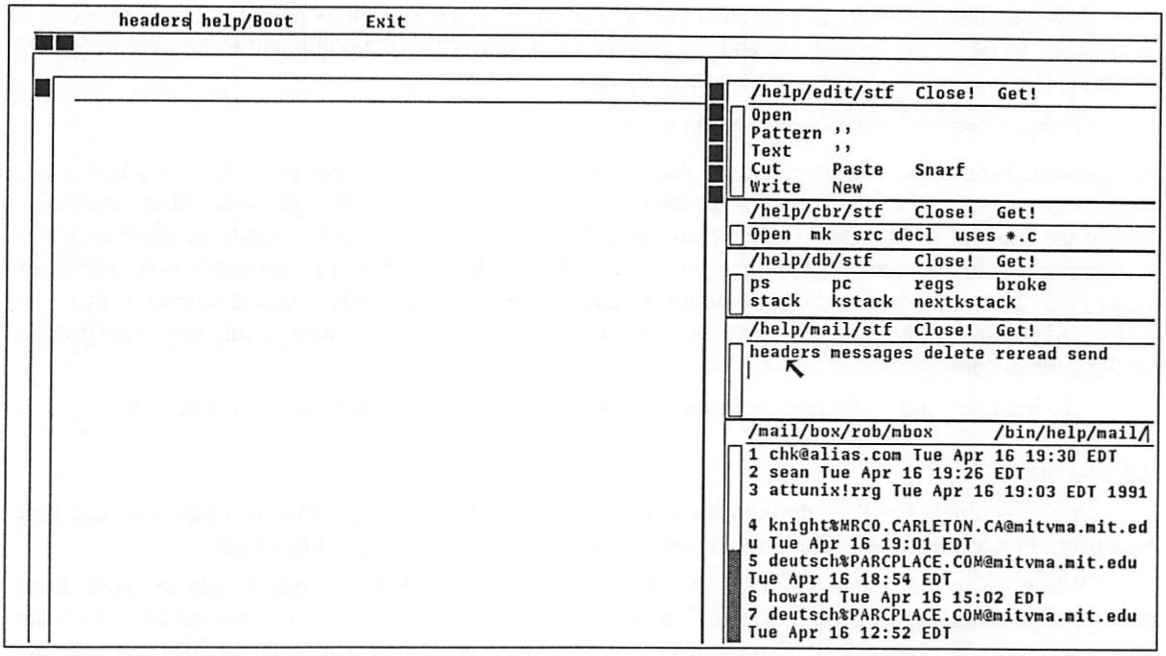


Figure 5: After executing `mail/headers`.

`Headers` creates a new window containing the headers of my mail messages, and labels it `/mail/box/rob/mbox`. I know Sean has sent me mail, so I point at the header of his mail (just

pointing with the left button anywhere in the header line will do) and click on `messages`.

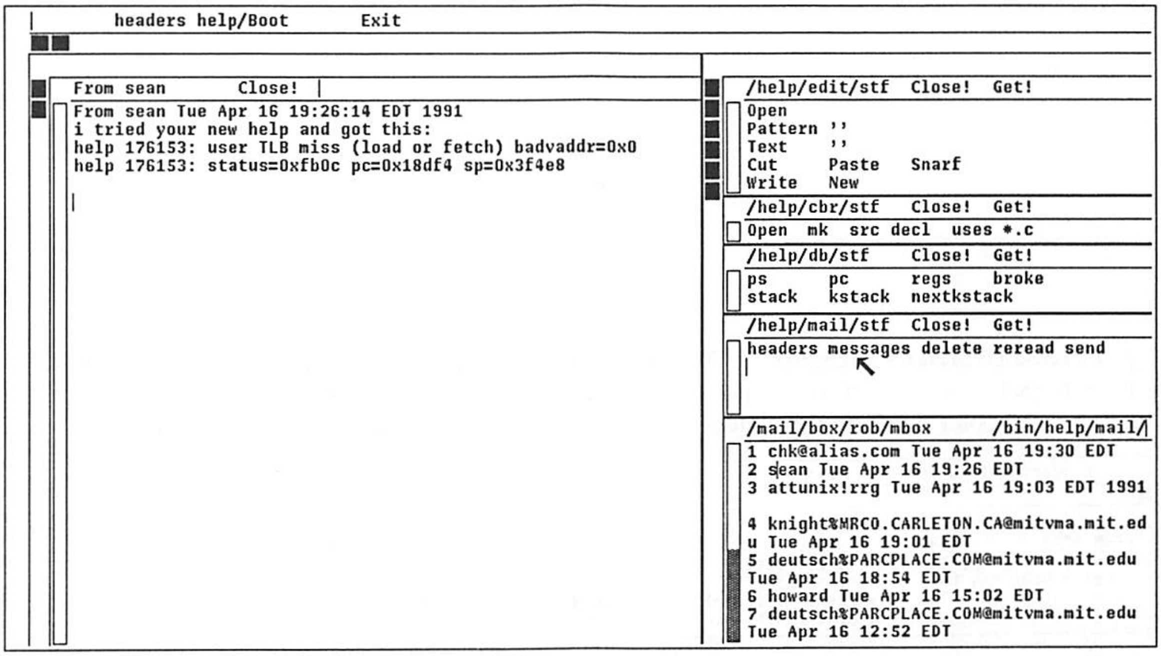


Figure 6: After applying `messages` to the header line of Sean's mail.

A new version of `help` has crashed and a broken process lies about waiting to be examined. (This is a property of Plan 9, not of `help`.) I point at the process number (I certainly shouldn't have to type it) and execute `stack` in the debugger tool, `/help/db/stf`. This pops up a window containing the traceback as reported by `adb` under the auspices of `/help/db/stack`.

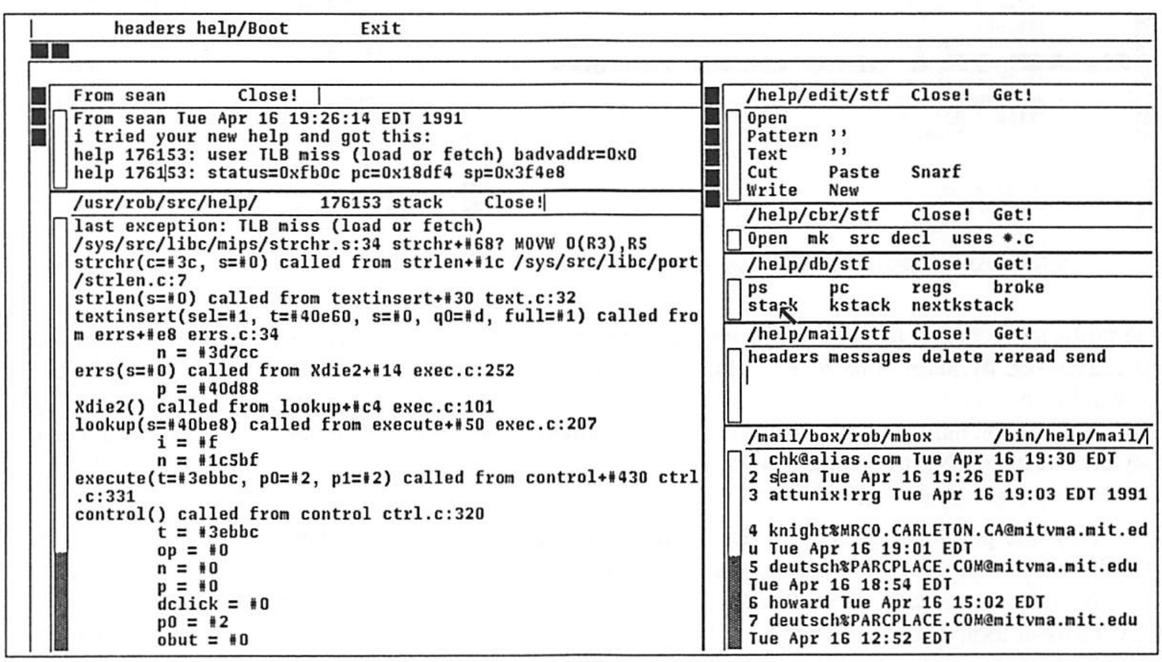


Figure 7: After applying `db/stack` to the broken process.

Notice that this new window has many file names in it. These are extracted from the symbol table of the broken program. I can look at the line (of assembly language) that died by pointing at the entry

/sys/src/libc/mips/strchr.s:34 and executing Open, but I'm sure the problem lies further up the call stack. The deepest routine in help is textinsert, which calls strlen on line 32 of the file text.c. I point at the identifying text in the stack window and execute Open to see the source.

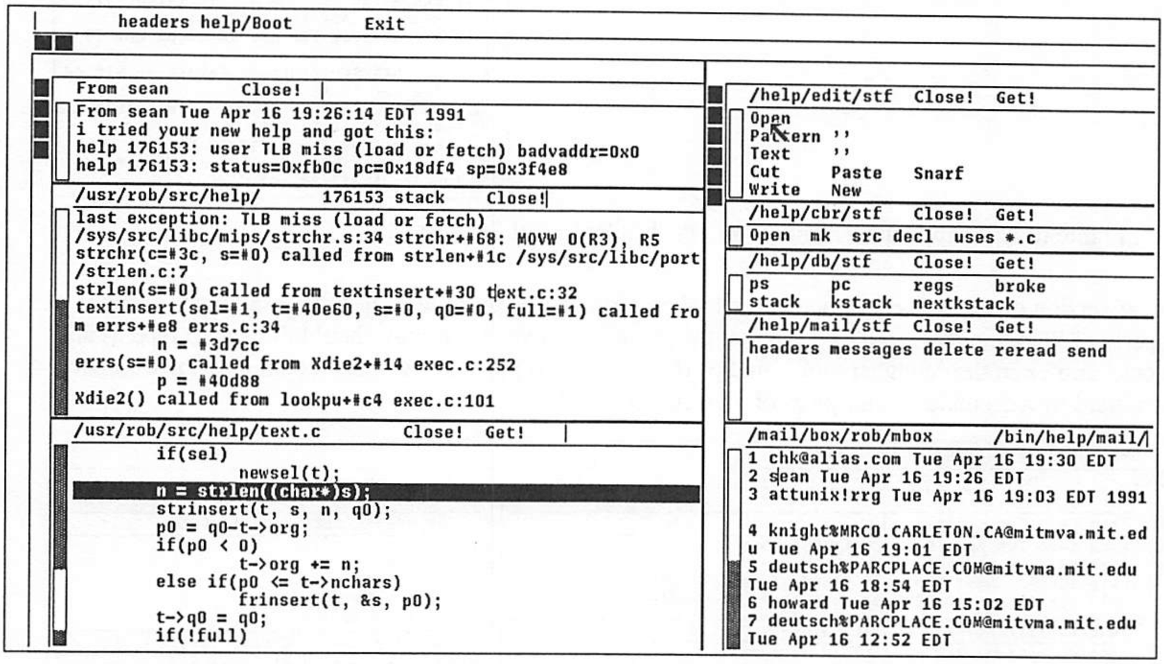


Figure 8: After Opening text.c at line 32.

The problem is coming to light: s, the argument to strlen, is zero, and was passed as an argument to textinsert by the routine errs, which apparently also got it as an argument from Xdie2. I close the window on text.c by hitting Close! in the tag of the window. By convention, commands ending in an exclamation mark take no arguments; they are window operations that apply to the window in which they are executed. Next I examine the source of the suspiciously named Xdie2 by pointing at the stack trace and Opening again. (See Figure 9.)

Now the problem gets harder. The argument passed to errs is a variable, n, that appears to be global. Who set it to zero? I can look at all the uses of the variable in the program by pointing at the variable in the source text and executing uses *.c by sweeping both 'words' with the middle button in the C browser tool, /help/cbr/stf. Uses creates a new window with all references to the variable n in the files /usr/rob/src/help/*.c indicated by file name and line number. The implementation of the C browser is described below; in a nutshell, it parses the C source to interpret the symbols dynamically. Compare this to running

```
grep n /usr/rob/src/help/*.c
```

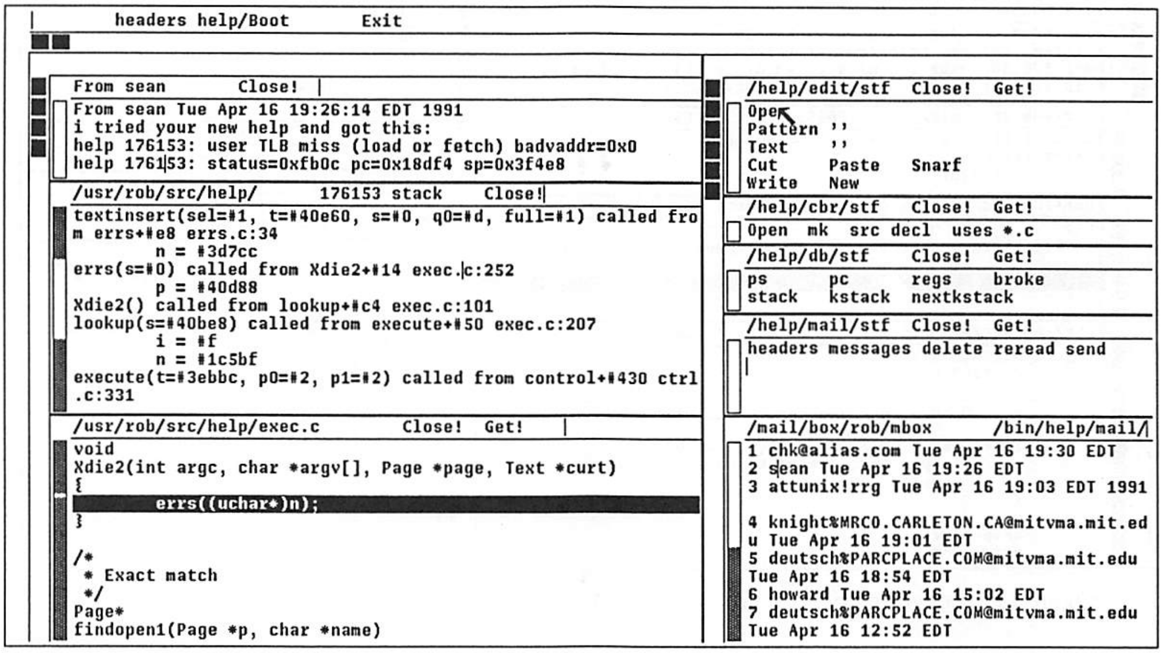


Figure 9: After `Opening` `exec.c` at line 252.

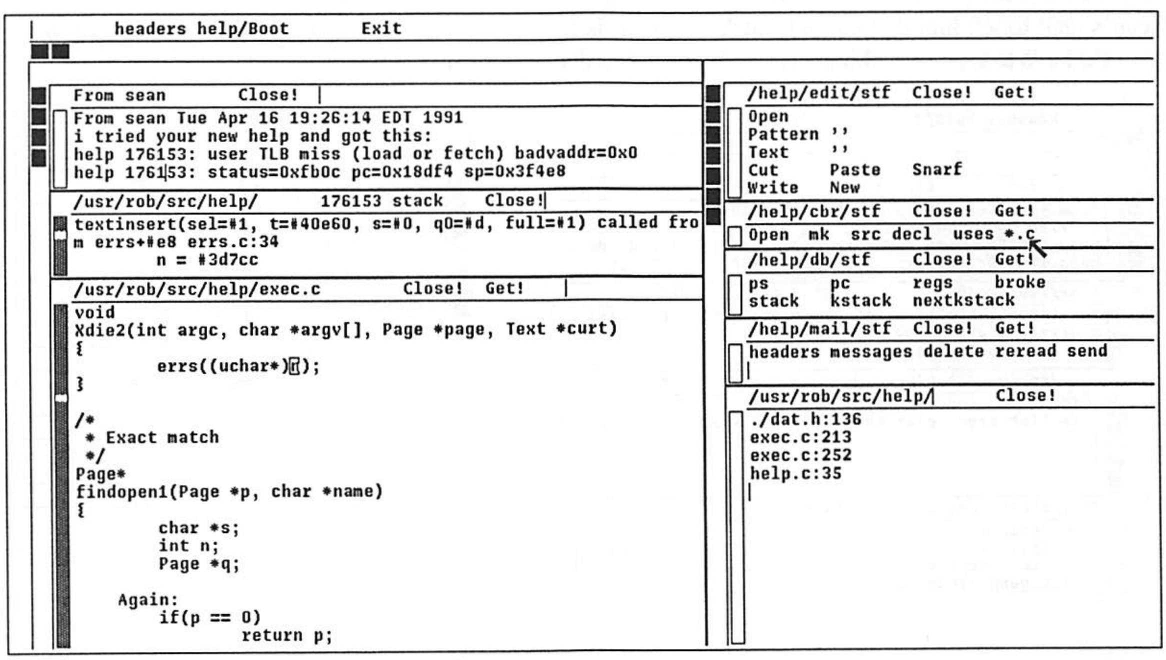


Figure 10: After finding all uses of `n`.

The first use is clearly the declaration in the header file. It looks like `help.c:35` should be an initialization. I `Open` `help.c` to that line and see that the variable is indeed initialized. (See Figure 11; a few lines off the top of the window is the opening declaration of `main()`.) Some other use of `n` must have cleared it. Line 252 of `exec.c` is the call; I know that's a read, not a write, of the variable. So I point to `exec.c:213` and execute `Open`.

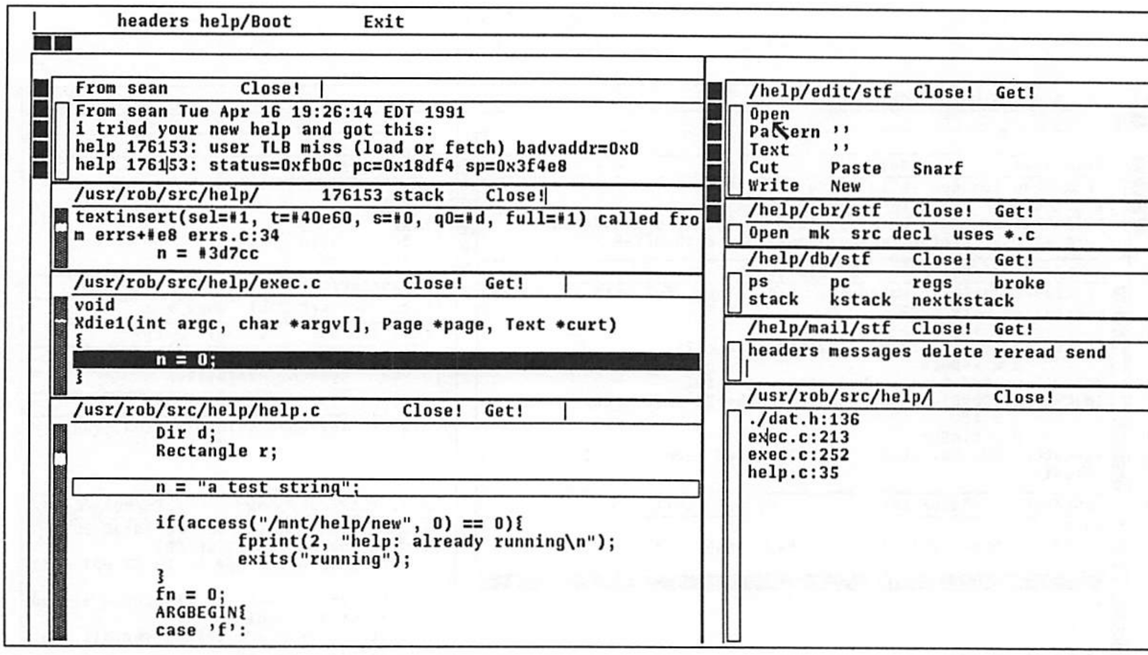


Figure 11: The writing of `n` on line `exec.c:213`.

Here is the jackpot of this contrived example. Sometime before `Xdie2` was executed, `Xdie1` cleared `n`. I use `Cut` to remove the offending line, write the file back out (the word `Put!` appears in the tag of a modified window) and then execute `mk` in `/help/cbr` to compile the program. I could now answer Sean's mail to tell him that the bug, such as it was, is fixed. I'll stop now, though, because to answer his mail I'd have to type something. Through this entire demo I haven't yet touched the keyboard.

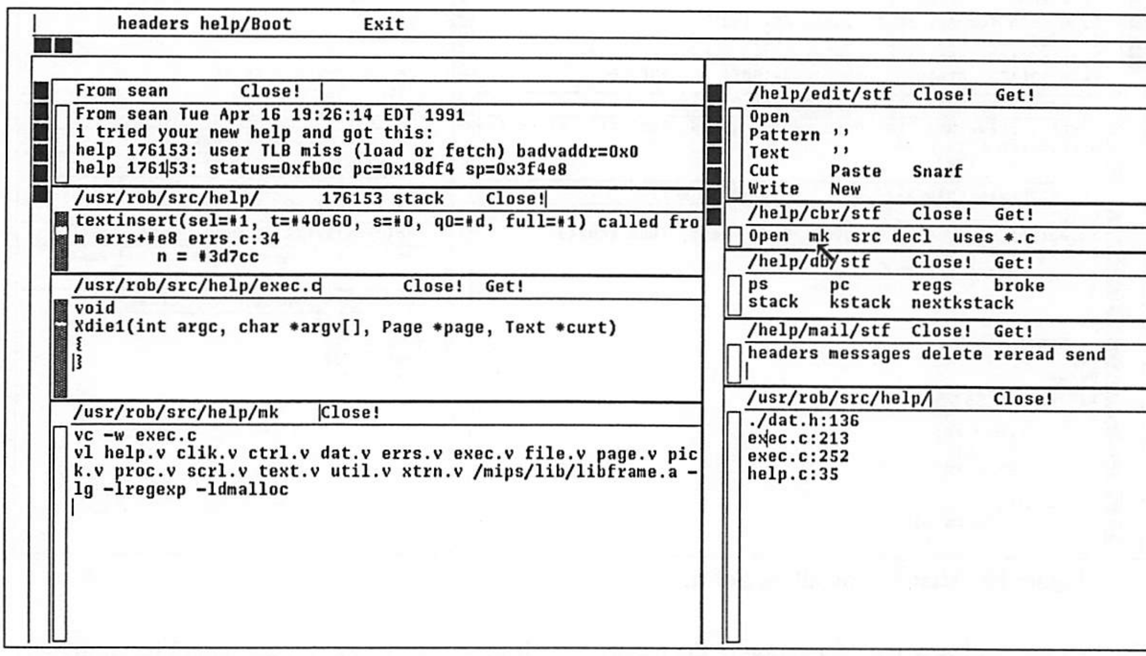


Figure 12: After the program is compiled.

This demonstration illustrates several things besides the general flavor of `help`. It is easy to work with files and commands in multiple directories. The rules by which `help` constructs file names from

context and, transitively, by which the utilities derive the context in which they execute simplify the management of programs and other systems constructed from scattered components. Also, the few common rules about text and file names allow a variety of applications to interact through a single user interface. For example, none of the tool programs has any code to interact directly with the keyboard or mouse. Instead `help` passes to an application the file and character offset of the mouse position. Using the interface described in the next section, the application can then examine the text in the window to see what the user is pointing at. These operations are easily encapsulated in simple shell scripts, an example of which is given in the next section.

The interface seen by programs

As in 8½, the Plan 9 window system [Pike91], `help` provides its client processes access to its structure by presenting a file service, although `help's` file structure is very different. Each `help` window is represented by a set of files stored in numbered directories. The number is a unique identifier, similar to UNIX process id's. Each directory contains files such as `tag` and `body`, which may be read to recover the contents of the corresponding subwindow, and `ctl`, to which may be written messages to effect changes such as insertion and deletion of text in contents of the window. The `help` directory is conventionally mounted at `/mnt/help`, so to copy the text in the body of window number 7 to a file, one may execute

```
cp /mnt/help/7/body file
```

To search for a text pattern,

```
grep pattern /mnt/help/7/body
```

An ASCII file `/mnt/help/index` may be examined to connect tag file names to window numbers. Each line of this file is a window number, a tab, and the first line of the tag.

To create a new window, a process just opens `/mnt/help/new/ctl`, which places the new window automatically on the screen near the current selected text, and may then read from that file the name of the window created, e.g. `/mnt/help/8`. The position and size of the new window is, as usual, chosen by `help`.

Another example

The directory `/help/cbr` contains the C browser we used above. One of the programs there is called `decl`; it finds the declaration of the variable marked by the selected text. Thus one points at a variable with the left button and then executes `decl` in the window for the file `/help/cbr/stf`. `Help` executes `/help/cbr/decl` using the context rules for the *executed* text and passes it the context (window number and location) of the *selected* text through an environment variable, `helpsel`.

`Decl` is a shell (rc) script. Here is the complete program:

```
eval `{help/parse -c}
x=`{cat /mnt/help/new/ctl}
{
        echo a
        echo $dir/'     Close!'
} | help/buf > /mnt/help/$x/ctl
{
        cpp $cppflags $file |
                help/rcc -w -g -i$id -n$line | sed 1q
} > /mnt/help/$x/bodyapp
```

The first line runs a small program, `help/parse`, that examines `$helpsel` and establishes another set of environment variables, `file`, `id`, and `line`, describing what the user is pointing at. The next creates a new window and sets `x` to its number. The first block writes the directory name to the tag line; the second runs the C preprocessor on the original source file (it should arguably be run on, say, `/mnt/help/8/body`) and passes the resulting text to a special version of the compiler. This compiler has no code generator; it parses the program and manages the symbol table, and when it sees the

declaration for the indicated identifier on the appropriate line of the file, it prints the file coordinates of that declaration. This appears on standard output, which is appended to the new window by writing to `/mnt/help/$x/bodyapp`. The user can then point at the output to direct `Open` to display the appropriate line in the source. (A future change to `help` will be to close this loop so the `Open` operation also happens automatically.) Thus with only three button clicks one may fetch to the screen the declaration, from whatever file in which it resides, the declaration of a variable, function, type, or any other C object.

A couple of observations about this example. First, `help` provided all the user interface. To turn a compiler into a browser involved spending a few hours stripping the code generator from the compiler and then writing a half dozen brief shell scripts to connect it up to the user interface for different browsing functions. Given another language, we would need only to modify the compiler to achieve the same result. *We would not need to write any user interface software.* Second, the resulting application is not a monolith. It is instead a small suite of tiny shell scripts that may be tuned or toyed with for other purposes or experiments.

Other applications are similarly designed. For example, the debugger interface, `/help/db`, is a directory of ten or so brief shell scripts, about a dozen lines each, that connect `adb` to `help`. `Adb` has a notoriously cryptic input language; the commands in `/help/db` package the most important functions of `adb` as easy-to-use operations that connect to the rest of the system while hiding the rebarbative syntax. People unfamiliar with `adb` can easily use `help's` interface to it to examine broken processes. Of course, this is hardly a full-featured debugger, but it was written in about an hour and illustrates the principle. It is a prototype, and `help` is an easy-to-program environment in which to build such test programs. A more sophisticated debugger could be assembled in a similar way, perhaps by leaving a debugging process resident in the background and having the `help` commands send it requests.

Discussion

`Help` is a research prototype that explores some ideas in user interface design. As an experiment it has been successful. When someone first begins to use `help`, the profusion of windows and the different ground rules for the user interface are disorienting. After a couple of hours, though, the system seems seductive, even natural. To return at that point to a more traditional environment is to see how much smoother `help` really is. Unfortunately, it is sometimes necessary to leave `help` because of its limitations.

The time has probably come to rewrite `help` with an eye to robustness and such mundane but important features as undo, multiple windows per file, the ability to handle large files gracefully, support for traditional shell windows, and syntax for shell-like functionality such as I/O redirection. Also, of course, the restriction to textual applications should be eliminated. `Help` has a couple of other major weaknesses. The first is that the heuristics for screen layout are not good enough. In the major example above, I rearranged the windows at many of the steps to keep things organized sensibly. On the other hand, the heuristics have not been worked on or even thought about much; there are probably a few simple ideas that could improve them dramatically.

The other weakness is that `help` does not exploit the Plan 9 environment as well as it could. The most obvious example is that the applications run on the same machine as `help` itself. This is probably easy to fix: `help` could run on the terminal and make an invisible call to the CPU server, sending requests to run applications to the remote shell-like process. This is similar to how `nmake` [Fowl90] runs its subprocesses, so the idea should be workable.

If imitation is the sincerest form of flattery, the designers of Oberon's user interface will (I hope) be honored by `help`. But Oberon has some aspects that made it difficult to adapt the user interface directly to UNIX-like systems such as Plan 9. The most important is that Oberon is a monolithic system constructed in and around a module-based language. An Oberon tool, for instance, is essentially just a listing of the entry points of a module. In retrospect, the mapping of this idea into commands in a UNIX directory may seem obvious, but it took a while to discover. Once it was found, the idea to use the directory name associated with a file or window as a context, analogous to the Oberon module, was a real jumping-off point. `Help` only begins to explore its ramifications.

Another of Oberon's difficulties is that it is a single-process system. When an application is running, all other activity — even mouse tracking — stops. It turned out to be easy to adapt the user interface to a multi-process system. `Help` may even be superior in this regard to traditional shells and window systems since it makes a clean separation between the text that executes a command and the result of this command. When windows are cheap and easy to use why not just create a window for every process? Also, `help's` structure as a Plan 9 file server makes the implementation of this sort of multiplexing straightforward.

One of the directions I would like to explore is that of compilation control. Running `make` in the appropriate directory is too pedestrian for an environment like this. Also, for complicated trees of source directories, the `makefiles` would need to be modified so the file names would couple well with `help's` way of working. `Make` and `help` don't function in similar ways. `Make` works by being told what target to build and looking at which files have been changed that are components of the target. What's needed for `help` is almost the opposite: a tool that, perhaps by examining the `index` file, sees what source files have been modified and builds the targets that depend on them. Such a program may be a simple variation of `make` — the information in the `makefile` would be the same — or it may be a whole new tool. Either way, it should be possible to tighten the binding between the compilation process and the editing of the source code; deciding what work to do by noticing file modification times is inelegant.

`Help` is similar to a hypertext system, but the connections between the components are not in the data — the contents of the windows — but rather in the way the system itself interprets the data. `Help` doesn't have the static quality of a hypertext system; when a new application is installed in `help` it combines multiplicatively with the other applications rather than additively.

The most important feature of `help` is that it provides a comfortable, easy-to-program user interface that all applications may share. It greatly simplifies the job of providing an interactive 'front end' for a new application. The ability to write such applications in the shell is a welcome change from the programming required for most window systems. `Help` is not a panacea, of course, but it does a good enough job to serve as a prototyping system that provides the user interface, freeing the programmer to concentrate on the job at hand, the construction of new software tools.

Acknowledgements

Sean Dorward wrote the mail tools and suggested many improvements to `help`. Doug Blewett, Tom Duff, Stu Feldman, Eric Grosse, Dennis Ritchie, and Howard Trickey made helpful comments on the paper. Judy Paone and Howard Trickey assisted in assembling the camera-ready copy.

References

[Fowl90] Glenn Fowler, "A Case for make", Softw. - Prac. and Exp., Vol 20 #S1, June 1990, pp. 30-46

[Haeb90] Paul Haeberli, "ConMan: A Visual Programming Language for Interactive Graphics", Comp. Graph., Vol 22 #4, Aug. 1988, pp. 103-110

[Oust90] John Ousterhout, "Tcl: An Embeddable Command Language", Proc. USENIX Winter 1990 Conf., pp. 133-146

[Pike90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, "Plan 9 from Bell Labs", Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 1-9

[Pike91] Rob Pike, "8½, the Plan 9 Window System", USENIX Summer Conf. Proc., Nashville, June, 1991, this volume

[Reis91] Martin Reiser, *The Oberon System,* Addison Wesley, New York, 1991

[Wirt89] N. Wirth and J. Gutknecht, "The Oberon System", Softw. - Prac. and Exp., Sep 1989, Vol 19 #9, pp 857-894

The Plan 9 system, including the window system and perhaps the help program, is being made available to universities on an as-is basis in source form. If you are interested, please contact:

Rob Pike
Bell Labs 2C524
Murray Hill NJ 07974
rob@research.att.com

Rob Pike is a Member of Technical Staff at AT&T Bell Laboratories in Murray Hill, New Jersey, where he has been since 1980, the same year he won the Olympic silver medal in Archery. In 1981 he wrote the first bitmap window system for UNIX systems, and has since written nine more. With Bart Locanthi he designed the Blit terminal; with Brian Kernighan he wrote *The Unix Programming Environment*. A shuttle mission nearly launched a gamma-ray telescope he designed. He is a Canadian citizen and has never written a program that uses cursor addressing.

Integrating Gesture Recognition and Direct Manipulation

Dean Rubine
Information Technology Center
Carnegie Mellon University
Dean.Rubine@cs.cmu.edu

Abstract

A gesture-based interface is one in which the user specifies commands by simple drawings, typically made with a mouse or stylus. A single intuitive gesture can simultaneously specify objects, an operation, and additional parameters, making gestures more powerful than the "clicks" and "drags" of traditional direct-manipulation interfaces. However, a problem with most gesture-based systems is that an entire gesture must be entered and the interaction completed before the system responds. Such a system makes it awkward to use gestures for operations that require continuous feedback.

GRANDMA, a tool for building gesture-based applications, overcomes this shortcoming through two methods of integrating gesturing and direct manipulation. First, GRANDMA allows views that respond to gesture and views that respond to clicks and drags (e.g. widgets) to coexist in the same interface. More interestingly, GRANDMA supports a new two-phase interaction technique, in which a gesture collection phase is immediately followed by a manipulation phase. In its simplest form, the phase transition is indicated by keeping the mouse still while continuing to hold the button down. Alternatively, the phase transition can occur once enough of the gesture has been seen to recognize it unambiguously. The latter method, called *eager recognition*, results in a smooth and natural interaction. In addition to describing how GRANDMA supports the integration of gesture and direct manipulation, this paper presents an algorithm for creating eager recognizers from example gestures.

1. Introduction

Gestures are hand-drawn strokes that are used to command computers. The canonical example is a proofreader's mark used for editing text [2, 4] shown in figure 1.

Gesture-based systems are similar to handwriting systems [26], in that both rely on pattern recognition for interpreting drawn symbols. Unlike a handwritten character, a single gesture indicates an operation, its operands, and additional parameters. For example, the gesture in figure 1 might be translated "move these characters to this location," with the referents clearly indicated by the gesture.

Generally, the end of the gesture must be indicated before the gesture is classified and command execution commences. In almost every gesture-based system to date, the gesture ends when the user

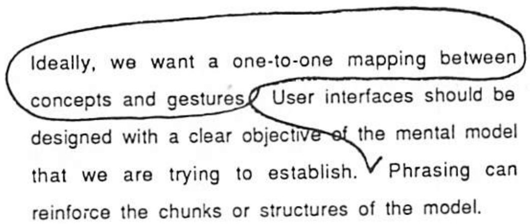


Figure 1: A move text gesture (from Buxton [2])

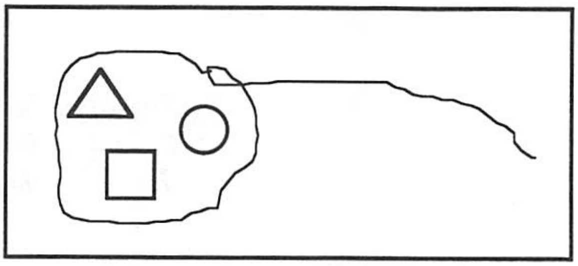


Figure 2: Moving objects in GEdit (from Kurtenbach and Buxton [14])

relaxes physically, e.g. by releasing the mouse button or lifting the stylus from the tablet. (Examples include IBM's Paper-Like Interface [11, 21], Coleman's proofreader's marks [4], MCC's HITS [10], and Buxton's char-rec note input tool [3].) The physical tension and relaxation of making a gesture correlates pleasantly with the mental tension and relaxation involved in performing a primitive task in the application [2]. However, since command execution begins only after the interaction ends, there is no opportunity for semantic feedback from the application *during* the interaction.

Referring to the move text gesture, Kurtenbach and Buxton [14] claim that application feedback is unnecessary during the interaction. They do admit that in the case of a drawing editor (figure 2) the lack of feedback hampers precise positioning. In the text case, they are probably correct that actually moving the text during the interaction is undesirable. What is desirable, I claim, is feedback in the form of a text cursor, dragged by the mouse but snapping [1] to legal destinations for the text. Such a cursor confirms that the gesture was indeed recognized correctly, and allows the user to be sure of the text's destination before committing to the operation by releasing the mouse button.

Direct manipulation [25] is an accepted paradigm for providing application feedback during mouse interactions. The goal of the present work is to combine gesturing with direct manipulation. One way this might be done is via modes: after a gesture is recognized, the following mouse interaction is interpreted as a manipulation rather than another gesture. A better way[1] is to interpret mouse interactions as gestures when they begin on certain objects, and otherwise as direct manipulation. This is done in GEdit [14]: a mouse press on a shape causes it to be dragged, while a mouse press over the background window is interpreted as gesture. An alternative would be to use one mouse button for gesturing and another for direct manipulation. While these techniques work, they may result in a primitive application task in the user's mental model ("create a rectangle of a given size and position") being serialized into multiple interactions ("create a rectangle" then "manipulate its size", then "manipulate its position"). This serialization undoes the correlation between physical and mental tension.

This paper advocates integrating gesture and direct manipulation in a single, two-phase interaction. The intent is to retain the intuitiveness of each interaction style, while combining their power.

[1]Modes, the "global variable" of user interfaces, are generally frowned upon.

The first phase of the interaction is *collection*, during which the points of the gesture are collected. Then the end of the gesture is indicated, the gesture classified, and the *manipulation* phase is entered. The classification of the gesture determines the operation to be performed. The operand and some parameters may also be determined at classification time. During the manipulation phase, additional parameters may be determined interactively, in the presence of application feedback.

GRANDMA (Gesture Recognizers Automated in a Novel Direct Manipulation Architecture) is a system I built for creating gesture-based applications [22, 23]. Written in Objective-C [5], GRANDMA runs on a DEC MicroVax II under MACH [27] and X10 [24]. In GRANDMA, the time of the transition from collection to manipulation is determined in one of three ways:

1. when the mouse button is released (in which case the manipulation phase is omitted),
2. by a timeout indicating that the user has not moved the mouse for 200 milliseconds, or
3. when enough of the gesture has been seen to unambiguously classify it.

The last alternative, termed *eager recognition*, results in smooth and graceful interactions. Citing my dissertation, Henry *et. al.* [9] describes hand-coded eager recognizers for a particular application.

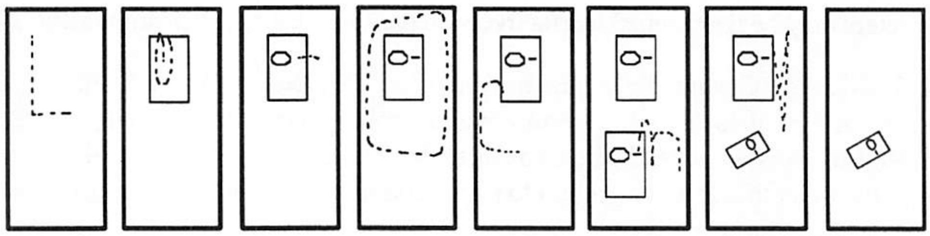
The present work focuses on *trainable* recognition, in which gesture recognizers, both eager and not, are built from example gestures. The discussion begins with GDP, a gesture-based drawing program which combines gesture and direct manipulation. The next section describes an algorithm for constructing eager recognizers from training examples. Performance measurements of the algorithm are covered in the following section. A concluding section summarizes the work and presents some future directions.

2. GDP: A gesture-based drawing program

GDP is a gesture-based drawing program based on (the non-gesture-based program) DP [7]. GDP is capable of producing drawings made with lines, rectangles, ellipses, and text. This section sketches GDP's operation from the user's perspective.

Figure 3 shows the effect of a sequence of GDP gestures. (Eager recognition has been turned off, so full gestures are shown.) The user presses the mouse button and enters the rectangle gesture and then stops, holding the button down. The gesture is recognized, and a rectangle is created with one endpoint at the start of the gesture, another endpoint at the current mouse location. The latter endpoint can then be dragged by the mouse: this enables the rectangle's size to be determined interactively.

The line and ellipse gestures work similarly. The group gesture generates a composite object out of the enclosed objects; additional objects may be added to the group by touching them during the manipulation phase. The copy gesture replicates an object, allowing it to be positioned during manipulation. The move gesture, not shown, works analogously. The initial point of the rotate-scale gesture determines the center of rotation; the final point (i.e. the mouse position when the gesture is recognized) determines a point (not necessarily on the object) that will be dragged around to interactively manipulate the object's size and orientation. The delete gesture deletes the object



Gesture:	Rectangle	Ellipse	Line	Group	Copy	Rotate-scale	Delete
Determined at recognition time:	Corner 1	Center	Endpt 1	Enclosed objects to group	Object to copy	Object Center of rotation Drag point	Object to delete
Determined by manipulation:	Corner 2	Size Eccentricity	Endpt 2	Touch other objects to add	Location of copy	Size Orientation	Touch additional objects to delete

Figure 3: Some GDP gestures and parameters (adapted from [22])

Gestures are shown with dotted lines. The effect of each gesture is shown in the panel to its right. Under each panel are listed those parameters that are determined at the time the gesture is recognized, and those that may be manipulated in the presence of application feedback.

at the gesture start. During the manipulation phase, any additional objects touched by the mouse cursor are also deleted.

In a modified version of GDP, the initial angle of the rectangle gesture determines the orientation of the rectangle (with respect to the horizontal). For this to work, the rectangle gesture was trained in multiple orientations. In the version shown here, only the "L" orientation was used in training. Also in the modified version, the length of the line gesture determines the thickness of the line. To keep things simple, the modified version was not used either to generate the figure or in the remainder of this paper. It is mentioned here to illustrate how gestural attributes may be mapped to application parameters.

Not shown is an edit gesture (which looks like "Σ"). This gesture brings up control points on an object. The control points do not themselves respond to gesture, but can be dragged around directly (scaling the object accordingly). This illustrates that systems built with GRANDMA can combine gesture and direct manipulation in the same interface.

Note that each gesture used in GDP is a single stroke. This is a limitation of GRANDMA's gesture recognition algorithm. Supporting only single stroke gestures reinforces the correlation between physical and mental tension, allows the use of short timeouts, and simplifies both non-eager gesture recognition and eager recognition. The major drawback is that many common marks (e.g. "X" and "—>") cannot be used as gestures by GRANDMA. A number of techniques exist for adapting single-stroke recognizers to multiple stroke recognition [8, 15], so perhaps GRANDMA's recognizer will extended this way in the future.

3. Support for gesture and direct manipulation in GRANDMA

The following summary of GRANDMA's architecture is intended to be sufficient for explaining how gesture and direct manipulation are integrated. Much detail has been glossed over; the interested reader is referred to [23] for the full story.

GRANDMA is a Model/View/Controller-like system [13]. In GRANDMA, models are application objects, views are objects responsible for displaying models, and event handlers deal with input directed at views. GRANDMA generalizes MVC by allowing a list of event handlers (rather than a single controller) to be associated with a view. Event handlers may be associated with view classes as well, and are inherited. Associating a handler with an entire class greatly improves efficiency, as a single handler is automatically shared by many objects.

3.1. Gesture and direct-manipulation in the same interface

Each class of event handler implements a particular kind of interaction technique. For example, the drag handler handles drag interactions, enabling entire objects (or parts of objects) to be dragged by the mouse. A gesture handler contains a classifier for a set of gestures, and handles both the collection and manipulation phases of the two-phase interaction. Thus, it is straightforward in GRANDMA to have some views respond to gesture while other respond to direct manipulation: simply associate gesture handlers with the former's class and drag handlers (or other direct-manipulation style handlers) with the latter's. Similarly, views of different classes may respond to different sets of gestures by associating each view class with a different gesture handler.

A single view (or view class) may respond to both gesture and direct manipulation (say, via different mouse buttons) by associating multiple handlers with the view. Each handler has a predicate that it uses to decide which events it will handle. It is simple to arrange for a handler to deal only with particular types of events (e.g. mouse down, mouse moved, mouse up) or only with events generated by a particular mouse button. The handlers associated with a particular view are queried in order whenever input is initiated at the view; any input ignored by one handler is propagated to the next.

3.2. Gesture and direct-manipulation in a two-phase interaction

As mentioned above, the gesture handler implements the two-phase interaction technique. Each instance of a gesture handler recognizes its own set of gestures, and can have its own semantics associated with each gesture. The handler is responsible for collecting and inking the gesture, determining when the phase transition occurs, classifying the gesture, and executing the gesture's semantics.

The gesture semantics consist of three expressions: `recog`, evaluated when the gesture is recognized (i.e. at the phase transition), `manip`, evaluated for each mouse point that arrives during the manipulation phase, and `done`, evaluated when the interaction ends (i.e. the mouse button is released). For example, the semantics of GDP's rectangle gesture are:

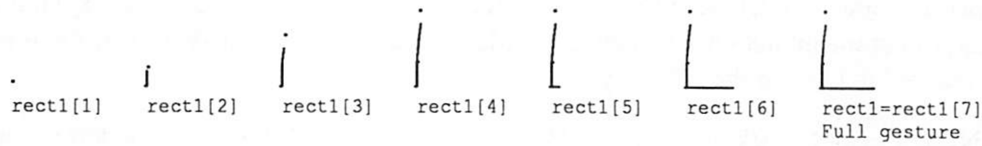


Figure 4: A full rectangle gesture and all its subgestures.

```
recog = [[view createRect]
               setEndpoint:0 x:<startX> y:<startY>];
manip = [recog setEndpoint:1 x:<currentX> y:<currentY>];
done = nil;
```

The syntax is that of Objective-C messages. The expressions are evaluated by a simple Objective-C message interpreter built into GRANDMA. During evaluation, the values of many gestural attributes are lazily bound to variables in the environment, and are thus available for use as parameters in application routines. In the above example, `view` refers to the object at which the gesture is directed, in this case the window in which GDP runs. This view is sent to message `createRect`, which returns a newly created rectangle. The attributes `<startX>` and `<startY>` refer to the initial point of the gesture; the newly created rectangle is sent a message making this point one corner of the rectangle. The rectangle is conveniently stored in the variable `recog` for use in the `manip` semantics. In response to each mouse point during the manipulation phase, the `manip` semantics makes the other corner of the rectangle `<currentX>`, `<currentY>`, thus implementing the interactive "rubberbanding" of the rectangle. The `done` expression is null in this case, as the processing was done by `manip`. There are many other attributes available to the semantics writer; see [22, 23] for details.

This section described how, using GRANDMA, gestures and direct manipulation may be combined in an interface by associating different handlers with different views (or with the same view but having different predicates). The two-phase interaction technique, in which gesture and direct-manipulation are combined in a single interaction, is implemented by the gesture handler class. Each gesture handler knows how to collect gestures, classify them as elements of the gesture set expected by the handler, and execute the corresponding gesture semantics.

4. Eager Recognition

As has been seen, gestures may be combined with direct manipulation to create a powerful two-phase interaction technique. This section focuses on how the transition between the phases may be made without any explicit indication from the user. Thus far the paper has concentrated (1) on the description of the two-phase interaction technique, (2) on GRANDMA, a system which supports the technique, and (3) on the use of the technique in GDP, an example application. The treatment in this section is at a lower level: here we are concerned with the pattern recognition technology that is used to implement eager recognition, a particular flavor of the two-phase interaction technique.

4.1. Gestures, subgestures, and full gestures

A gesture is defined as a sequence of points. Denote the number of points in a gesture g as $|g|$, and the particular points as $g_p = (x_p, y_p, t_p)$, $0 \le p < |g|$. The triple (x, y, t) represents a two-dimensional mouse point (x, y) that arrived at time t. (The actual content of the points turns out to be largely irrelevant for the eager recognition algorithm presented below, and is only given here for concreteness.)

The i^{th} *subgesture* of g, denoted $g[i]$, is defined as a gesture consisting of the first i points of g. Thus, $g[i]_p = g_p$ and $|g[i]| = i$. The subgesture $g[i]$ is simply a prefix of g, and is undefined when $i > |g|$. The term *full gesture* is used when it is necessary to distinguish the full gesture g from its proper subgestures $g[i]$ for $i < |g|$ (see figure 4).

4.2. Statistical single-stroke gesture recognition

We are given a set of C gesture classes, and a number of (full) example gestures of each class, $g_{\hat{c}e}$, $0 \le c < C$, $0 \le e < E_{\hat{c}}$, where $E_{\hat{c}}$ is the number of training examples of class c. In GDP, $C = 11$ (the classes are line, rectangle, ellipse, group, text, delete, edit, move, rotate-scale, copy, and dot) and typically we train with 15 examples of each class, i.e. $E_{\hat{c}} = 15$.

The (non-eager) gesture recognition problem is stated as follows: given an input gesture g, determine the class c to which g belongs, i.e. the class whose training examples $g_{\hat{c}e}$ are most like g. (Some of the vagueness here can be eliminated by assuming each class c has a certain probability distribution over the space of gestures, that the training examples of each class were drawn according to that class's distribution, and that the recognition problem is to choose the class c whose distribution, as revealed by its training examples, is the most likely to have produced g.) A classifier $\mathcal{C}$ is a function that attempts to map g to its class c: $c = \mathcal{C}(g)$. As $\mathcal{C}$ is trained on the full gestures $g_{\hat{c}e}$, it is referred to here as a *full* classifier.

The field of pattern recognition in general [6], and on-line handwriting recognition in particular [26], offers many suggestions on how to compute $\mathcal{C}$ given training examples $g_{\hat{c}e}$. Popular methods include the Ledeen recognizer [18] and connectionist models (i.e. neural networks) [8, 10]. Curiously, many gesture researchers [3, 9, 12, 14, 17] choose to hand-code $\mathcal{C}$ for their particular application, rather than attempt to create it from training examples. Lipscomb [15] presents a method tailored to the demands of gesture recognition (rather than handwriting), as does the current author[22, 23].

My method of classifying single-stroke gestures, called *statistical gesture recognition*, works by representing a gesture g by a vector of (currently twelve) features $\mathbf{f}$. Each feature has the property that it can be updated in constant time per mouse point, thus arbitrarily large gestures can be handled. Classification is done via linear discrimination: each class has a linear evaluation function (including a constant term) that is applied to the features, and the class with the maximum evaluation is chosen as the value of $\mathcal{C}(g)$. Training is also efficient, as there is a closed form expression (optimal given some normality assumptions on the distribution of the feature vectors of a class) for determining the evaluation functions from the training data.

There are two more properties of the single-stroke classifier that are exploited by the eager recognition algorithm below. The first is the ability to handle differing costs of misclassification:

simply by adjusting the constant terms of the evaluation functions, it is possible to bias the classifier away from certain classes. This is useful when mistakenly choosing a certain class is a grave error, while mistakenly *not* choosing that class is a minor inconvenience. The other property used below is a side effect of computing a classifier. Theoretically, the computed classifier works by creating a distance metric (the Mahalanobis distance[6]), and the chosen class of a feature vector is simply the class whose mean is closest to the given feature vector under this metric. As will be seen, the distance metric is also used in the construction of eager recognizers.

4.3. The Ambiguous/Unambiguous Classifier

In order to implement eager recognition, a module is needed that can answer the question "has enough of the gesture being entered been seen so that it may be unambiguously classified?" If the gesture seen so far is considered to be a subgesture $g[i]$ of some full gesture g that we have yet to see, we can ask the question this way: "are we reasonably sure that $\mathcal{C}(g[i]) = \mathcal{C}(g)$?" The goal is to design a function $\mathcal{D}$ (for "done") that answers this question: $\mathcal{D}(g[i]) =$ `false` if $g[i]$ is ambiguous (i.e. there might be two full gestures of different classes both of which have $g[i]$ as a subgesture), and $\mathcal{D}(g[i]) =$ `true` if $g[i]$ is unambiguous (meaning all full gestures that might have $g[i]$ as a subgesture are of the same class).

Given $\mathcal{D}$, eager recognition works as follows: Each time a new mouse point arrives it is appended to the gesture being collected, and $\mathcal{D}$ is applied to this gesture. As long as $\mathcal{D}$ returns `false` we iterate and collect the next point. Once $\mathcal{D}$ return `true` the collected gesture is passed to $\mathcal{C}$ whose result is return and the manipulation phase entered.

The problem of eager recognition is thus to produce $\mathcal{D}$ from the given training examples. The insight here is to view this as a classification problem: classify a given subgesture as an ambiguous or unambiguous gesture prefix. The recognition techniques developed for single-path recognition (and discussed in the previous section) are used to build the ambiguous/unambiguous classifier (AUC). $\mathcal{D}$ returns `true` if and only if the AUC classifies the subgesture as unambiguous.

4.4. Complete and incomplete subgestures

Once the idea of using the AUC to generate $\mathcal{D}$ is accepted, it is necessary to produce data to train the AUC. Since the purpose of the AUC is to classify subgestures as ambiguous or unambiguous, the training data must be subgestures that are labeled as ambiguous or unambiguous.

We may use the full classifier $\mathcal{C}$ to generate a first approximation to these two sets (ambiguous and unambiguous). For each example gesture of class c, $g = g_e^{\hat{c}}$, some subgestures $g[i]$ will be classified correctly by the full classifier $\mathcal{C}$, while others likely will not. A subgesture $g[i]$ is termed *complete* with respect to gesture g, if, for all $j, i \leq j < |g|, \mathcal{C}(g[j]) = \mathcal{C}(g)$. The remaining subgestures of g are *incomplete*. A complete subgesture is one which is classified correctly by the full classifier, and all larger subgestures (of the same gesture) are also classified correctly.

Figure 5 shows examples of two gestures classes, U and D. Both start with a horizontal segment, but U gestures end with an upward segment, while D gestures end with a downward segment. In this simple example, it is clear that the subgestures which include only the horizontal segment are

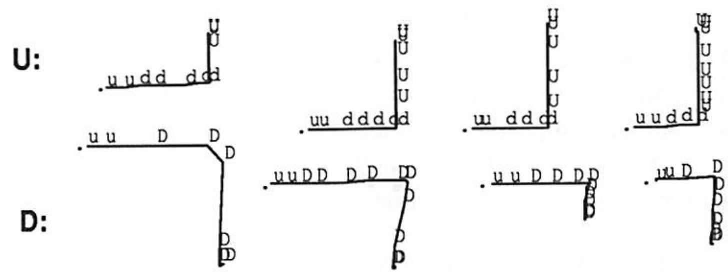


Figure 5: Incomplete and complete subgestures of U and D

The character indicates the classification (by the full classifier) of each subgesture. Uppercase characters indicate complete subgestures, meaning that the subgesture and all larger subgestures are correctly classified. Note that along the horizontal segment (where the subgestures are ambiguous) some subgestures are complete while others are not.

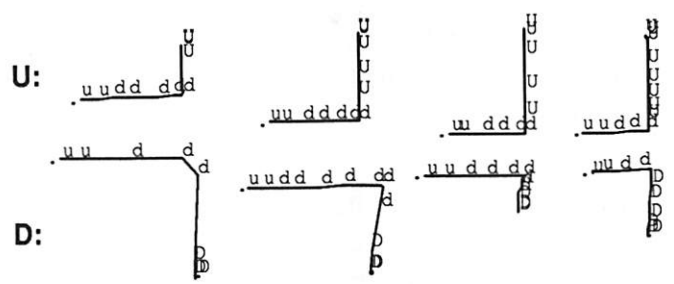


Figure 6: Accidentally complete subgestures have been moved

Comparing this to figure 5 it can be seen that the subgestures along the horizontal segment of the D *gestures have been made incomplete. Unlike before, after this step all ambiguous subgestures are incomplete.*

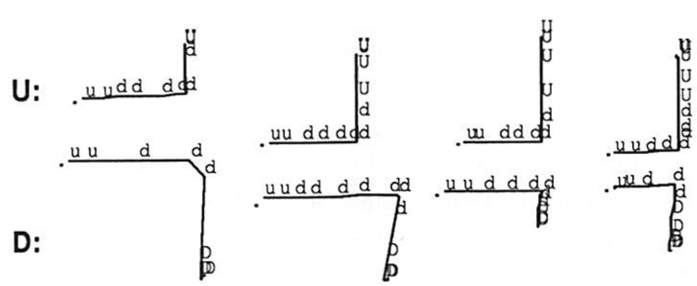


Figure 7: Classification of subgestures of U and D

This shows the results of running the AUC on the training examples. As can been seen, the AUC performs conservatively, never indicating that a subgestures is unambiguous when it is not, but sometimes indicating ambiguity of an unambiguous subgesture.

ambiguous, but subgestures which include the corner are unambiguous. In the figure, each point in the gesture is labeled with a character indicating the classification by C of the subgesture which ends at the point. An uppercase label indicates a complete subgesture, lowercase an incomplete subgesture. Notice that incomplete subgestures are all ambiguous, all unambiguous subgestures are complete, but there are complete subgestures that are ambiguous (along the horizontal segment of the D examples). These subgestures are termed *accidentally* complete since they happened to be classified correctly even though they are ambiguous.

It turns out that even if it is possible to completely determine which subgestures are ambiguous and which are not, using the training methods referred to in section 4.2 to produce a single-stroke recognizer to discriminate between the two classes ambiguous and unambiguous does not work very well. This is because the training methods assume that the distribution of feature vectors within a class is approximately multivariate Gaussian. The distribution of feature vectors within the set of unambiguous subgestures will likely be wildly non-Gaussian, since the member subgestures are drawn from many different gesture classes. For example, in the figure the unambiguous U subgestures are very different than the unambiguous D gestures, so there will be a bimodal distribution of feature vectors in the unambiguous set. Thus, a linear discriminator will not be adequate to discriminate between two classes ambiguous and unambiguous subgestures. What must be done is to turn this two-class problem (ambiguous or unambiguous) into a multi-class problem. This is done by breaking up the ambiguous subgestures into multiple classes, each of which has an approximately normal distribution. The unambiguous subgestures must be similarly partitioned.

To do this, instead of partitioning the example subgestures into just two sets (complete and incomplete), they are partioned into $2C$ sets. These sets are named C-c and I-c for each gesture class c. A complete subgesture $g[i]$ is placed in the class C-c, where $c = C(g[i]) = C(g)$. An incomplete subgesture $g[i]$ is placed in the class I-c, where $c = C(g[i])$ (and it is likely that $c \neq C(g)$). The sets I-c are termed incomplete sets, and the sets C-c, complete sets. Note that the class in each set's name refers to the full classifier's classification of the set's elements. In the case of incomplete subgestures, this is likely not the class of the example gesture of which the subgesture is a prefix. In figure 5 each lowercase letter names a set of incomplete subgestures, while each uppercase letter names a set of complete subgestures.

4.5. Moving Accidentally Complete Subgestures

The next step in generating the training data is to move any accidentally complete subgestures into incomplete classes. Intuitively, it is possible to identify accidentally complete subgestures because they will be similar to some incomplete subgestures (for example, in figure 5 the subgestures along the horizontal segment are all similar even though some are complete and others are incomplete.) A threshold applied to the Mahalanobis distance metric mentioned in section 4.2 may be used to test for this similarity.

To do so, the distance of each subgesture $g[i]$ in each complete set to the mean of each incomplete set is measured. If $g[i]$ is sufficiently close to one of the incomplete sets, it is removed from its complete set, and placed in the closest incomplete set. In this manner, an example subgesture that was accidentally considered complete (such as a right stroke of a D gesture) is grouped together with the other incomplete right strokes (class I-D in this case).

Quantifying exactly what is meant by "sufficiently close" turns out to be rather difficult. The threshold on the distance metric is computed as follows: The distance of the mean of each full gesture class to the mean of each incomplete subgesture class is computed, and the minimum found. However, distances less than another threshold are not included in the minimum calculation to avoid trouble when an incomplete subgesture looks like a full gesture of a different class. (This is the case if, in addition to U and D, there is a third gesture class consisting simply of a right stroke.) The threshold used is 50% of that minimum.

The complete subgestures of a full gesture are tested for accidental completeness from largest (the full gesture) to smallest. Once a subgesture is determined to be accidentally complete, it and the remaining (smaller) complete subgestures are moved to the appropriate incomplete classes.

Figure 6 shows the classes of the subgestures in the example after the accidentally complete subgestures have been moved. Note that now the incomplete subgestures (lowercase labels) are all ambiguous.

4.6. Create and tweak the AUC

Now that there is training data containing C complete classes (indicating unambiguous subgestures), and C incomplete classes (indicating ambiguous subgestures), it is a simple matter to run the single-stroke training algorithm (section 4.2) to create a classifier to discriminate between these $2C$ classes. This classifier will be used to compute the function $\mathcal{D}$ as follows: if this classifier places a subgesture s in any incomplete class, $\mathcal{D}(s)$ = `false`, otherwise the s is judged to be in one of the complete classes, in which case $\mathcal{D}(s)$ = `true`.

It is very important that subgestures not be judged unambiguous wrongly. This is a case where the cost of misclassification is unequal between classes: a subgesture erroneously classified ambiguous will merely cause the recognition not to be as eager as it could be, whereas a subgesture erroneously classified unambiguous will very likely result in the gesture recognizer misclassifying the gesture (since it has not seen enough of it to classify it unambiguously). To avoid this, the constant terms of the evaluation function of the incomplete classes i are incremented by a small amount. The increment is chosen to bias the classifier so that it believes that ambiguous gestures are five times more likely than unambiguous gestures. In this way, it is much more likely to choose an ambiguous class when unsure.

Each incomplete subgesture is then tested on the new classifier. Any time such a subgesture is classified as belonging to a complete set (a serious mistake), the constant term of the evaluation function corresponding to the complete set is adjusted automatically (by just enough plus a little more) to keep this from happening.

Figure 7 shows the classification by the final classifier of the subgestures in the example. A larger example of eager recognizers is presented in the next section.

4.7. Summarizing the eager recognition training algorithm

While the details are fairly involved, the idea behind the eager recognition technology is straightforward. The basic problem is to determine if enough of a gesture has been seen to classify it

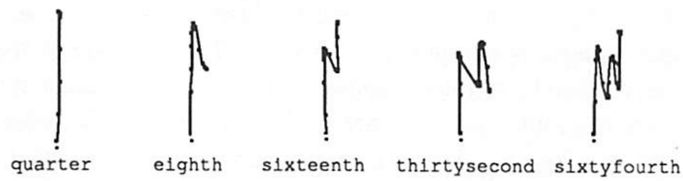


Figure 8: A set of gestures not amenable to eager recognition
Because all but the last gesture is approximately a subgesture of the one to its right, these gestures would always be considered ambiguous by the eager recognizer, and thus would never be eagerly recognized. The period indicates the first point of each gesture.

unambiguously. This determination is itself a classification problem to which the same trainable gesture recognition technology may be applied. The main hurdle is to produce data to train this classifier to discriminate between ambiguous and unambiguous subgestures. This is done by running the full classifier on every subgesture of the original training examples. Any subgesture classified differently than the full gesture from which it arose is considered ambiguous; also ambiguous are those subgestures that happen to be classified the same as their full gestures but are similar to subgestures already considered to be ambiguous. For safety, after being trained to discriminate between ambiguous and unambiguous subgestures, the new classifier is conservatively biased toward classifying subgestures as ambiguous.

5. Evaluating Eager Recognition

How well the eager recognition algorithm works depends on a number of factors, the most critical being the gesture set itself. It is very easy to design a gesture set that does not lend itself well to eager recognition; for example, there would be almost no benefit trying to use eager recognition on Buxton's note gestures [3] (figure 8). This is because the note gestures for longer notes are subgestures of the note gestures for shorter notes, and thus would always be considered ambiguous by the eager recognizer.

In order to determine how well the eager recognition algorithm works, an eager recognizer was created to classify the eight gestures classes shown in 9. Each class named for the direction of its two segments, e.g. "ur" means "up, right." Each of these gestures is ambiguous along its initial segment, and becomes unambiguous once the corner is turned and the second segment begun.

The eager recognizer was trained with ten examples of each of the eight classes, and tested on thirty examples of each class. The figure shows ten of the thirty test examples for each class, and includes all the examples that were misclassified.

Two comparisons are of interest for the gesture set: the eager recognition rate versus the recognition rate of the full classifier, and the eagerness of the recognizer versus the maximum possible eagerness. The eager recognizer classified 97.0% of the gestures correctly, compared to 99.2% correct for the full classifier. Most of the eager recognizer's errors were due to a corner looping 270 degrees rather than being a sharp 90 degrees, so it appeared to the eager recognizer the second stroke was going in the opposite direction than intended. In the figure "E" indicates a gesture misclassified by the eager recognizer, and "F" indicates a misclassification by the full classifier.

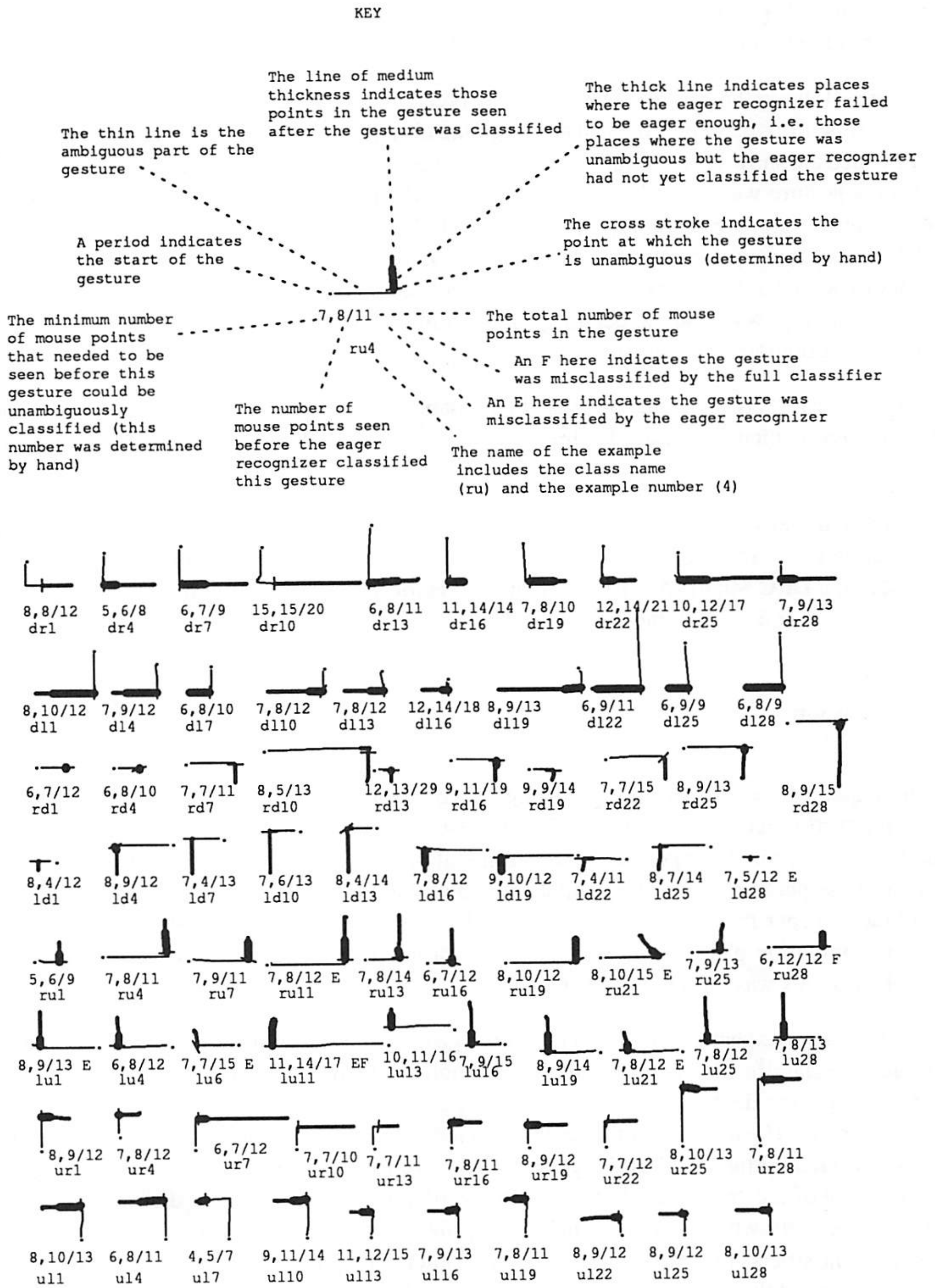


Figure 9: The performance of the eager recognizer on easily understood data

On the average, the eager recognizer examined 67.9% of the mouse points of each gesture before deciding the gesture was unambiguous. By hand I determined for each gesture the number of mouse points from the start through the corner turn, and concluded that on the average 59.4% of the mouse points of each gesture needed to be seen before the gesture could be unambiguously classified. The parts of each gesture at which unambiguous classification could have occurred but did not are indicated in the figure by thick lines.

Figure 10 shows the performance of the eager recognizer on GDP gestures. The eager recognizer was trained with 10 examples of each of 11 gesture classes, and tested on 30 examples of each class, five of which are shown in the figure. The GDP gesture set was slightly altered to increase eagerness: the group gesture was trained clockwise because when it was counterclockwise it prevented the copy gesture from ever being eagerly recognized. For the GDP gestures, the full classifier had a 99.7% correct recognition rate as compared with 93.5% for the eager recognizer. On the average 60.5% of each gesture was examined by the eager recognizer before classification occurred. For this set no attempt was made to determine the minimum average gesture percentage that needed to seen for unambiguous classification.

From these tests we can conclude that the trainable eager recognition algorithm performs acceptably but there is plenty of room for improvement, both in the recognition rate and the amount of eagerness.

Computationally, eager recognition is quite tractable on modest hardware. A fixed amount of computation needs to occur on each mouse point: first the feature vector must be updated (taking 0.5 msec on a DEC MicroVAX II), and then the vector must be classified by the AUC (taking 0.27 msec per class, or 6 msec in the case of GDP).

6. Conclusion

In this paper I have shown how gesturing and direct manipulation can be combined in a two-phase interaction technique that exhibits the best qualities of both. These include the ability to specify an operation, the operands, and additional parameters with a single, intuitive stroke, with some of those parameters being manipulated directly in the presence of application feedback. The technique of eager recognition allows a smooth transition between the gesture collection and the direct manipulation phases of the interaction. An algorithm for creating eager recognizers from example gestures was presented and evaluated.

There is an unexpected benefit of combining gesture and direct manipulation in a single interaction: gesture classification is often simpler and more accurate. Consider the "move text" gesture in figure 1. Selecting the text to move is a circling gesture that will not vary too much each time the gesture is made. However, after the text is selected the gesture continues and the destination of the text is indicated by the "tail" of the gesture. The size and shape of this tail will vary greatly with each instance of the "move text" gesture. This variation makes the gesture difficult to recognize in general, especially when using a trainable recognizer. Perhaps this is why many researchers hand code their classifiers. In any case, in a two-phase interaction the tail is no longer part of the gesture, but instead part of the manipulation. Trainable recognition techniques will be much more successful on the remaining prefix.

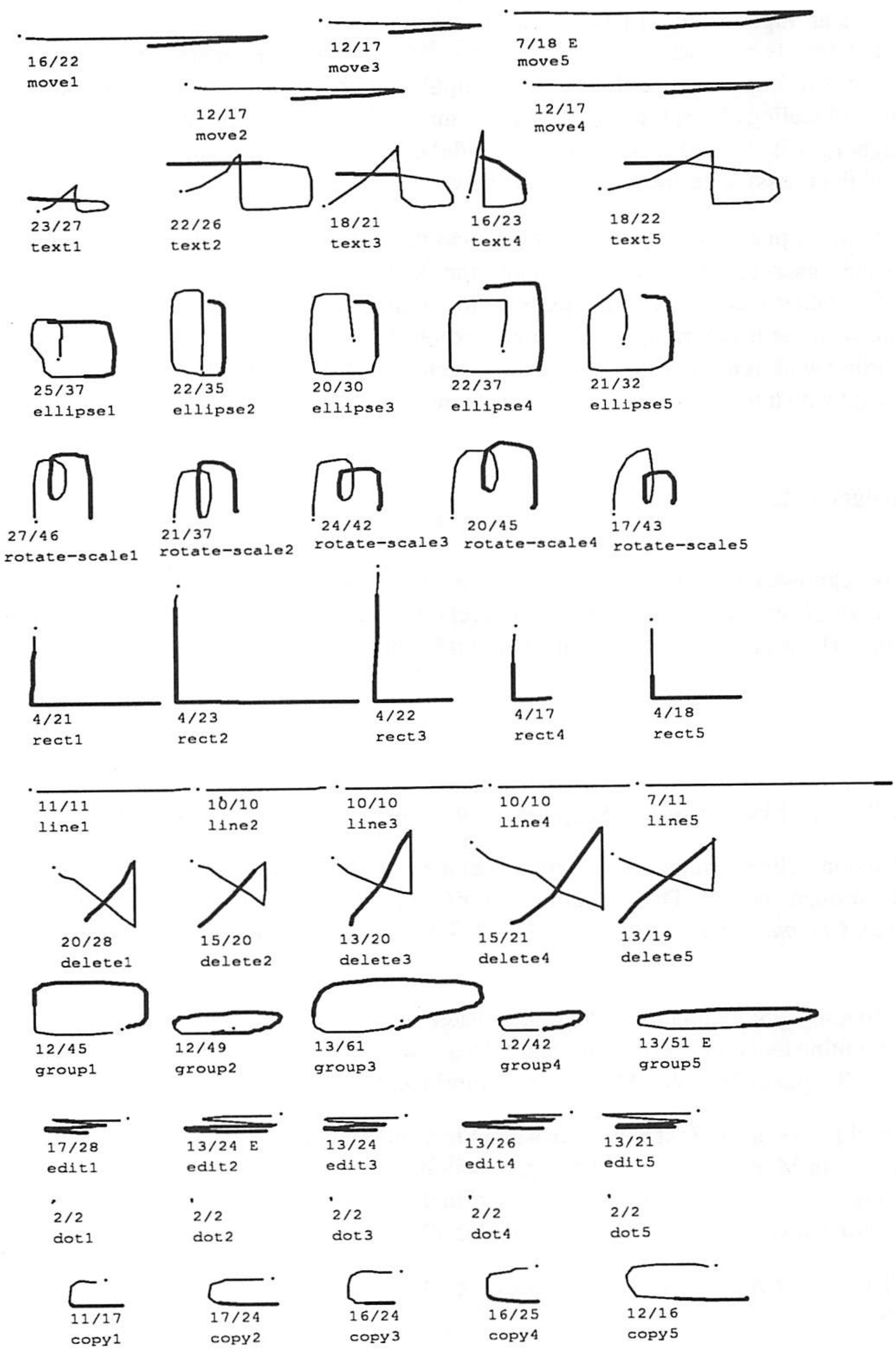


Figure 10: The performance of the eager recognizer on GDP gestures

The transitions from thin to thick lines indicates where eager recognition occurred.

The two-phase interaction technique is also applicable to multi-path gestures. Using the Sensor Frame [16] as an input device, I have implemented a drawing program based on multiple finger gestures. The results have been quite encouraging. For example, the translate-rotate-scale gesture is made with two fingers, which during the manipulation phase allow for simultaneous rotation, translation, and scaling of graphic objects. Even some single finger gestures allow additional fingers to be brought into the field of view during manipulation, thus allowing additional parameters (such as color and thickness) to be specified interactively.

In the future, I plan to incorporate gestures (and the two-phase interaction) into some existing object-oriented user-interface toolkits, notably the Andrew Toolkit[20] and the NeXT Application Kit[19]. Other extensions including handling multi-stroke gestures, and integrating gesture recognition with the handwriting recognition used on the notebook computers now beginning to appear. Further work is needed to utilize devices, such as the DataGlove[28], which have no explicit signaling with which to indicate the start of a gesture.

Acknowledgements

I gratefully acknowledge CMU's School of Computer Science for supporting this work as part of my dissertation effort, and CMU's Information Technology Center for supporting me in the creation of this paper. Thanks also to Roger Dannenberg for his helpful comments.

References

[1] Eric Bier and Maureen Stone. Snap-dragging. *Computer Graphics*, 20(4):233–240, 1986.

[2] W. Buxton. There's more to interaction than meets the eye: Some issues in manual input. In D.A. Norman and S.W. Draper, editors, *User Centered Systems Design: New Perspectives on Human-Computer Interaction*, pages 319–337. Lawrence Erlbaum Associates, Hillsdale, N.J., 1986.

[3] W. Buxton, R. Sniderman, W. Reeves, S. Patel, and R. Baecker. The evolution of the SSSP score-editing tools. In Curtis Roads and John Strawn, editors, *Foundations of Computer Music*, chapter 22, pages 387–392. MIT Press, Cambridge, Mass., 1985.

[4] Michael L. Coleman. Text editing on a graphic display device using hand-drawn proofreader's symbols. In M. Faiman and J. Nievergelt, editors, *Pertinent Concepts in Computer Graphics, Proceedings of the Second University of Illinois Conference on Computer Graphics*, pages 283–290. University of Illinois Press, Urbana, Chicago, London, 1969.

[5] Brad J. Cox. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1986.

[6] R. Duda and P. Hart. *Pattern Classification and Scene Analysis.* Wiley Interscience, 1973.

[7] Dario Giuse. DP command set. Technical Report CMU-RI-TR-82-11, Carnegie Mellon University Robotics Institute, October 1982.

[8] I. Guyon, P. Albrecht, Y. Le Cun, J. Denker, and W. Hubbard. Design of a neural network character recognizer for a touch terminal. *Pattern Recognition*, forthcoming.

[9] T.R. Henry, S.E. Hudson, and G.L. Newell. Integrating gesture and snapping into a user interface toolkit. In *UIST '90*, pages 112–122. ACM, 1990.

[10] J. Hollan, E. Rich, W. Hill, D. Wroblewski, W. Wilner, K. Wittenberg, J. Grudin, and Members of the Human Interface Laboratory. An introduction to HITS: Human interface tool suite. Technical Report ACA-HI-406-88, Microelectronics and Computer Technology Corporation, Austin, Texas, 1988.

[11] IBM. The Paper-Like Interface. In *CHI '89 Techical Video Program: Interface Technology*, volume Issue 47 of *SIGGRAPH Video Review*. ACM, 1989.

[12] Joonki Kim. Gesture recognition by feature analysis. Technical Report RC12472, IBM Research, December 1986.

[13] Glenn E. Krasner and Stephen T. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, August 1988.

[14] G. Kurtenbach and W. Buxton. GEdit: A test bed for editing by contiguous gestures. to be published in SIGCHI Bulletin, 1990.

[15] James S. Lipscomb. A trainable gesture recognizer. *Pattern Recognition*, 1991. Also available as IBM Tech Report RC 16448 (#73078).

[16] P. McAvinney. Telltale gestures. *Byte*, 15(7):237–240, July 1990.

[17] Margaret R. Minsky. Manipulating simulated objects with real-world gestures using a force and position sensitive screen. *Computer Graphics*, 18(3):195–203, July 1984.

[18] W.M. Newman and R.F. Sproull. *Principles of Interactive Computer Graphics*. McGraw-Hill, 1979.

[19] NeXT. *The NeXT System Reference Manual*. NeXT, Inc., 1989.

[20] A.J. Palay, W.J. Hansen, M.L. Kazar, M. Sherman, M.G. Wadlow, T.P. Neuendorffer, Z. Stern, M. Bader, and T. Peters. The Andrew toolkit: An overview. In *Proceedings of the USENIX Technical Conference*, pages 11–23, Dallas, February 1988.

[21] James R. Rhyne and Catherine G. Wolf. Gestural interfaces for information processing applications. Technical Report RC12179, IBM T.J. Watson Research Center, IBM Corporation, P.O. Box 218, Yorktown Heights, NY 10598, September 1986.

[22] Dean Rubine. Specifying gestures by example. In *SIGGRAPH 91*. ACM, 1991.

[23] Dean Rubine. *The Automatic Recognition of Gestures*. PhD thesis, School of Computer Science, Carnegie Mellon University, forthcoming, 1991.

[24] R.W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2), April 1986.

[25] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, pages 57–62, August 1983.

[26] C.C. Tappert. On-line handwriting recognition - a survey. Technical Report RC 14045, IBM T.J. Watson Research Center, August 1987.

[27] A. Tevanian. MACH: A basis for future UNIX development. Technical Report CMU-CS-87-139, Carnegie Mellon University Computer Science Dept., Pittsburgh, PA, 1987.

[28] T. Zimmerman, J. Lanier, C. Blanchard, S. Bryson, and Y. Harvill. A hand gesture interface device. *CHI+GI*, pages 189–192, 1987.

Biography

Dean Rubine is currently research faculty at Carnegie Mellon's Information Technology Center. He has a B.S. and M.S. in Electrical Engineering and Computer Science from MIT and is really close to completing his Ph.D. in Carnegie Mellon's School of Computer Science. His research interests include gesture-based systems, audio digital signal processing, user interfaces, pattern recognition, real-time control, integrated multimedia, and computer music.

Activity Server: You can run but you can't hide

Sanjay Manandhar

MIT Media Laboratory
Cambridge, MA 02139 USA
sanjay@media-lab.mit.edu

Abstract

The activity server is a software utility that provides high-level information on what a user or a number of users are doing by combining data inputs from various sources. These sources are: 1) the *finger server*, 2) the *phone server*, and 3) the *location server*. The finger server provides information on users' activities or lack thereof on hosts within a local network; the phone server abstracts information from call progress of users' digital phones whereas the location server abstracts information from active badge sightings. Responses from all these unobtrusive data gathering sources are put together to form a composite model of the activities of users. Information may be collected only from or on behalf of participating users; each user within the user community can determine how much of his own information the activity server may receive. The main attributes of the activity server are: 1) it maintains history of all the participating users, 2) it synchronizes among the many, possibly conflicting, pieces of information from three sources, 3) it distinguishes interdependence of activities among users, and 4) it provides a high-level abstraction about users' activities.

1 Introduction

People are creatures of habit. The variety of activities a person is involved in over the course of a work day is limited. Furthermore, these activities may recur at some frequency from day to day. Some earlier work goes as far as to say that everyday activity is wholly routine [1]. Although a random slice of human activity is not deterministic, recurring activities in an office environment can be modeled informally. A server that will help build a model of activities in an office environment is proposed. This server, the "activity server," and the various auxiliary clients will be described in later sections.

The activity server solicits information from three other servers, the finger server, the phone server and the location server. (For clarity the latter three servers are called Listeners and unless explicit, the "server" shall refer to the activity server whereas the "clients" shall refer to the clients of the activity server). The state maintained by the activity server helps build a dynamic model of office activities so that the cumulative results and histories of activities may provide useful information to the user himself and other members of his work place.

2 Related Work

Work in traditional artificial intelligence (AI) has concentrated on plan recognition where it is assumed that there is a precise plan that a user activity will follow [4,7]. Many researchers working in interdisciplinary fields have proposed models varying in scope, complexity, and computational formality. For instance, in a technical paper from Xerox PARC, many dynamic models at micro and macro levels were provided [3]. Some newer work argues against planned models since it is very difficult to allow for unpredictable and unanticipated circumstances [1,2,9]. Although not a user modeling effort, the activity server uses a dynamic model that can infer the state of users and adapt continuously. Most of the user modeling work in AI and cognitive science research, including dialog systems [6], expert systems and student modeling have focused on systems that monitor user activities, and react to them. The activity server only monitors and builds an activity model; however, clients of the activity server that benefit from the user model can execute (react) accordingly. In addition, unlike many user modeling efforts that attempts to enhance user-system interaction or understanding, the activity server focuses on augmenting user-user interaction with the help and coordination of many discrete systems.

In the following sections the motivation behind building a system like the activity server and its overall architecture will be discussed. Subsequently, the internals of the Listeners as well as that of the activity server will be presented, followed by a discussion section.

3 Motivation

The activity server is designed with a view to improve interaction of co-workers, given that richer forms of communications like meeting in person, talking over the telephone and electronic mail, etc., are not always possible. The activity server is intended to answer questions like, "Can I have a meeting now with colleagues A, B and C?" or "What is the breakdown of time spent on meetings today?" or "Where is colleague X?" or "Who came by my office while I was away?" Such everyday questions constantly arise in an office environment and require active participation from the solicitor of the information. It is highly desirable to have instant answers or at least intelligent inferences to such questions.

An Example

Let's take a simple example from above: Where is colleague X? To answer this question, the activity server queries its three Listeners. The following are possible answers that it receives:

> **Phone server:** X is not on the (default) phone.
> **Location server:** Badge X is in its (default) office; it has been there for the last 10 hours.
> **Finger Server:** Idle time on the machine in her office is zero minutes (connected to terminal d0).

The activity server draws up conclusions about the location of colleague X from each Listener response and assigns a confidence level to each. From the phone server alone, it is clear that X is not in her office. From the response of the location server, X is, in fact, in her office but has not moved for 10 hours. The latter piece of information undermines the plausible conclusion that she is in her office. These are conflicting conclusions. But the finger server responds that the idle time on her (default) machine is nil, i.e., she is active. However, she is not active on the console (located in her office), but rather on terminal d0. Since all dN (N is an integer value) terminals are open for dialin access, colleague X has dialed in. Hence, the activity server will return with the conclusion that colleague X is not around, she has dialed in remotely. From the example above, some of the key attributes that motivated the activity server can be illustrated:

1) *History*: The server maintains history on its data. This is important in order to be able to infer the user's activities from his past actions.

2) *Multiple sources*: The server will draw upon its Listeners to get multiple, possibly conflicting, views on real world activities.

3) *Multi-party*: The activity server monitors many users simultaneously. Every member of the group may affect the state of other members of the group.

4) *High-level abstraction*: The activity server can assemble many pieces of discrete information, which by themselves are of little value but cumulatively allow abstraction of intelligent, high-level inferences of the users' activities.

4 Overview of the system

As mentioned earlier, the activity server adopts a client-server model. However, the activity server itself is a client to three other servers, the finger server, the location server and the phone server, which provide asynchronous events that help build a model of user activities.

Events from the Listeners are funneled through a common event handler which does some housekeeping such as timestamping and executing database lookups (see Figure 1). The most important of the housekeeping duties, however, is the updating of internal state. Further, a set of rules is applied to resolve conflicting Listener conclusions or emphasize conclusions in agreement. Hence, this module can be considered an arbiter and manager of the various Listeners. The high level inferences made by the rules are saved and are made avail

able to the the clients via the client library.

5 The Listeners

Listeners are the indispensable information-gathering servers that the activity server relies on. They are independent and service their own pool of clients; the activity server is a special client because it connects to all the Listeners simultaneously. Each Listener is self-sufficient and is oblivious of other Listeners; only the activity server, which collects data from all the Listeners, has the global view which allows it to make more complex inferences that span more than any one Listener's domain. What follows is a more complete description of the Listeners.

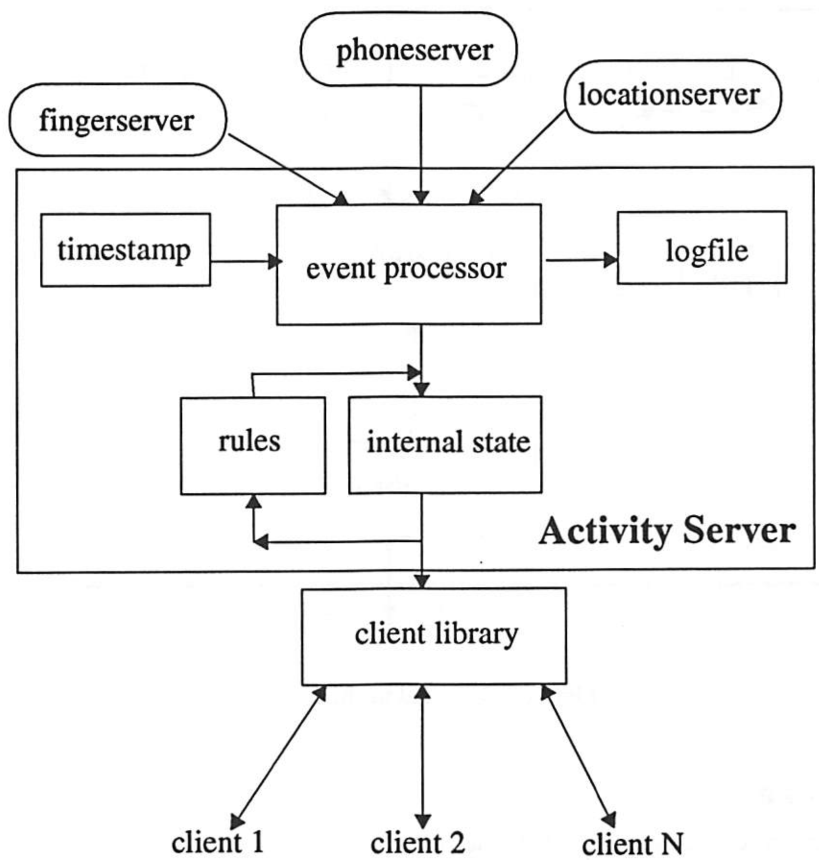


Figure 1. Overall system diagram of the activity server

5.1 The Finger Server

The finger server gathers idle times of users on a number of machines. This server depends on the finger daemon that runs on many machines. The finger daemon keeps state (idle times in minutes) of all users on that machine. The finger server queries finger daemons and collects and maintains information on many machines (Figure 2). This allows the finger server to compare information across the entire group of machines it is monitoring. For instance, if a user is logged in on more than one machine, it can tell where the user is most active, whether he is logged in remotely and from where he is logged in. In addition, the finger server can send asynchronous events (called alerts) to clients that express interest in them.

5.1.1 Architecture

The architecture of the finger server is based on the data objects it maintains. The finger server consists of doubly linked lists of three data objects (hosts, users and clients). The rationale behind the linked list is so that hosts, people, and clients are all dynamic data that may be added and deleted with ease. In addition, the basic linked list operations are the same for all the objects. Since the finger server is designed to keep state on a limited number of hosts, users and clients, data representation using linked lists is adequate; for a larger object domain, other optimal forms of data representation (e.g., hash tables) could be used.

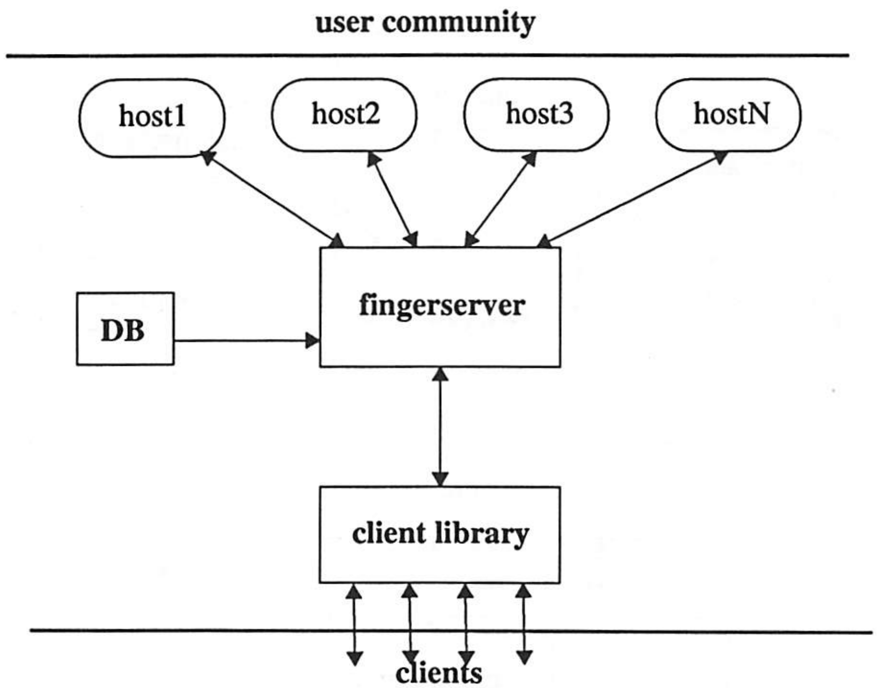


Figure 2. General architecture of the finger server

5.1.2 Mechanism

The finger server can be considered a client to the finger daemons while it is the server to its own clients. The finger server polls the list of machines that it keeps track of at certain intervals. It gets the list of machines and users to keep track of from a pre-defined text file which is checked periodically to see if has been updated. The results of the pollings are saved in the three linked lists mentioned above; queries from clients are serviced by accessing data from these linked lists.

5.1.3 Polling

A timer is registered so that the finger server can poll every time the the timer goes off. Typically the timer is set for 1.5 minutes. Note that the granularity of the finger daemon is in minutes so it is not useful to set the timer for less than a minute. Some machines may be polled less frequently if so desired. For instance, machines with many users or machines at a remote site may trade up-to-the-minute updating for better system performance.

It is imperative that the polling rely not only on the timer but also on the internal state of the polling session. It is unnecessary to query for finger information from a particular host a second time if the preceding poll has

not returned. If a socket to the finger daemon is opened at each polling interval, machines that take more than the poll cycle time to return from the poll may have, over time, multiple sockets open simultaneously.

5.1.4 Read Callbacks

At the time of polling, if the finger socket to a machine is currently not open, it is opened and a read callback is registered; the polling then moves on to the next machine - the server does not wait and block to read from a socket. Hence, the receipt of data is fully asynchronous. When data is available at a socket, its descriptor disambiguates the machine. The data is then read and the finger daemon closes the socket. Although it is a streams connection, the finger daemon keeps the socket open only as long as is necessary; that is, it is not possible to reserve a socket communication line for future polling.

5.1.5 Finger Information

Information returned by the finger daemon is machine-specific. So far, machines that use Unix, Ultrix or Genera (an OS for Symbolics Lisp machines) are supported. Typically the information received will have at least the username, hostname and the idle time; most Unix systems also report the terminal. All these pieces of data update the user and host data structures. After each host reports, its corresponding host data object and its internal linked list of users are updated and any changes are immediately reported to the clients that requested asynchronous notification. Reporting changes to the user data structure needs to wait until most machines have returned from the poll so that comparison of the most recent activity can be made and reported.

While idle times indicate the user's activity or lack thereof, the terminal name is useful in locating the user's physical location. If the user is connected to the console ("co") terminal of a machine and has a small or zero idle time, it is evident that no matter where else the user is logged on, she is physically located where the console of that machine is. On the other hand, if the user is on "dN" (N is an integer number) terminal, it is clear that the user has dialed in.

5.1.6 Protocols

There are two levels of finger server protocol. The lower level protocol is at the byte stream level; the second, a C interface, based on the byte stream level, is also available. Once a socket connection to the server is established, simple, case-insensitive, ASCII commands may be sent via the connection. The server uses the same connection to send its replies. This human-readable interface allows for understanding of commands and responses with a minimum experience with the system. A brief mode can be set so that non-human clients may get the important data easily. Figure 3 shows a sample session with the finger server using telnet. (User commands are shown in italics). The first command, help, gives a flavor for the kinds of functionalities that the finger server supports.

(toll: /u2/sanjay/due) telnet toll fingerserver
Connected to toll. Escape character is '^]'.
help
The text based commands and arguments are case insensitive.

?	Same as HELP
ASYNC	Commence asynchronous transmisions
BYE	Exit from the server
GET-HOST-INFO (host)	Get info on all users on HOST
GET-ALL-HOST-INFO	Get info on all users on all hosts
HELP	Print this help file
LIST-HOSTS	Get the names of all hosts the server knows about
LIST-PEOPLE	Get the names of all users the server knows about
LOCATE (person)	Find out where PERSON was logged on
LOCATE-ALL	Find out where everyone is logged on
QUIT	Same as BYE
RESET	Revert to default state (reset modes and requests)

SET-MODE (SHORT I LONG)	Set short or long output format
STATUS	Print out your personal requests and modes
SYNC	Stop asynchronous transmitions
USER-ON-HOST (user) (host)	Find out if USER is on HOST
TRACK (person)	Request asynchronous reporting on PERSON
TRACK-ALL	Request asynchronous reporting on all users and hosts
TRACK-ALL-HOSTS	Request asynchronous reporting on all hosts
TRACK-ALL-PEOPLE	Request asynchronous reporting on all users
TRACK-HOST (host)	Request asynchronous reporting on HOST
UNTRACK (person)	Cancel asynchronous reporting on PERSON
UNTRACK-ALL	Cancel asynchronous reporting on all users and hosts
UNTRACK-ALL-HOSTS	Cancel asynchronous reporting on all hosts
UNTRACK-ALL-PEOPLE	Cancel asynchronous reporting on all users
UNTRACK-HOST (host)	Cancel asynchronous reporting on HOST
VERSION	Current version of the finger server

locate barons
[PERSON barons] on [HOST leggett, TTY co] with [IDLE 3 min] at 03:21:02 PM

set-mode short
OKAY

get-host-info leggett
barons leggett co 6 03:23:21 PM

bye
Goodbye.
Connection closed by foreign host.

Figure 3. A sample session with the finger server

5.1.7 Alerts

Asynchronous reporting of changes in user or host status are of four kinds: login, logout, dormant and idle. All user data objects have their parent pointer pointing to the host object . After the poll to a host returns, the finger server updates all the user objects on that host and timestamps every object that is updated. In addition, all user objects maintain the idle and old idle times after the current and previous updates, respectively (reset value for both is -1). Given these premises, the algorithms that mark state transitions are relatively simple (see Figure 4). Note that IDLE_THRESHOLD is the ceiling of idle time up to which a user is considered active at a terminal. In the current implementation this user-defined value is 5 minutes.

State	Transition conditions	
LOGOUT:	**if**	user_timestamp > host_timestamp
LOGIN:	**if**	user_idle_old == -1
	and	user_idle > -1
DORMANT:	**if**	user_idle_old > IDLE_THRESHOLD
	and	user_idle > IDLE_THRESHOLD
IDLE:	**if**	user_idle_old > IDLE_THRESHOLD
	and	user_idle > IDLE_THRESHOLD

Figure 4. Simple algorithms for alerts

5.1.8 Shortcomings

There are some shortcomings which are inherent in the finger daemon while others were introduced by the finger server itself.

1. It is possible not to "see" quick logins and logouts on a machine. Since the polling cycle executes every few minutes, a login-logout pair on a machine by a particular user may not be reported by the finger daemon. Likewise, if a user exceeds the IDLE_THRESHOLD for a few seconds but then hits a key, this DORMANT state of up to 59 seconds is not reported. The granularity of the finger daemon is in minutes up to 59 minutes, then in hours and minutes up to 9 hours and 59 minutes, then in hours up to 23 hours and then in days. On the other hand more precise granularity may not be truly useful.

2. Activity in some programs are not noticeable to the finger daemon. For instance, activity solely in the gnuemacs editor program will increment the idle time as if the user was away from the terminal.

3. The finger daemon reports only the idle time, it does not report what program may be running. There are other Unix programs and daemons that monitor execution of programs at any particular time but their services were not used for a number of reasons.

- Not all machines run these daemons (rwho, w, rusers, etc.) but almost all machines keep the finger daemons running. (Some sites disable even the finger daemon for security reasons). Hence, finger-server can remain very general and modest in its requirements.
- Many of the other daemons run by mutual broadcasts and receives. This can put a severe burden on network and machine performance.
- Idle time history alone can be fairly useful.

4. One of the shortcomings of the finger server is that there is a latency in asynchronous reporting. Should there be a race condition between two hosts reporting back to the finger server, the updates of the second host on the queue will be delayed by the time it takes the first host to dump all its data and for the finger server to update all its data structures. Typically the latency is in the order of a few seconds. However, for machines with many users logged on, the reporting of everyone's information, and its subsequent parsing can take tens of seconds.

5.1.9 Optimizations

Querying the finger daemon to find a limited number of users' idle times is a very costly operation. It hurts network and machine performance. The finger daemon checks the /etc/hosts file and figures out usernames, real names, etc. Possible optimizations are: 1) provide options to the finger daemon that will return only the desired information. Unfortunately these options are not available on all machine types. If latency becomes critical, different requests can be sent to different machine types depending on what they support. 2) A modified finger daemon can be implemented that will cache much of the routine information. 3) A less frequent polling with interpolations between pollings (if necessary) can alleviate latencies.

5.2 The Phone Server

The phone server is another Listener that the activity server relies on to get phone activities. It monitors ISDN protocol that is exchanged between the 5ESS ISDN switch and the telephone sets to get the events of all telephone sets within its jurisdiction (see Figure 5). Much phone state information such as onhook, offhook, incoming call, dialing, held, etc. can be received from the phone server. For the purposes of the activity server, however, only onhook and offhook information are critical; to a lesser extent, the incoming call (caller number) may be useful if the calling and the called parties are both among the users the activity server is interested in.

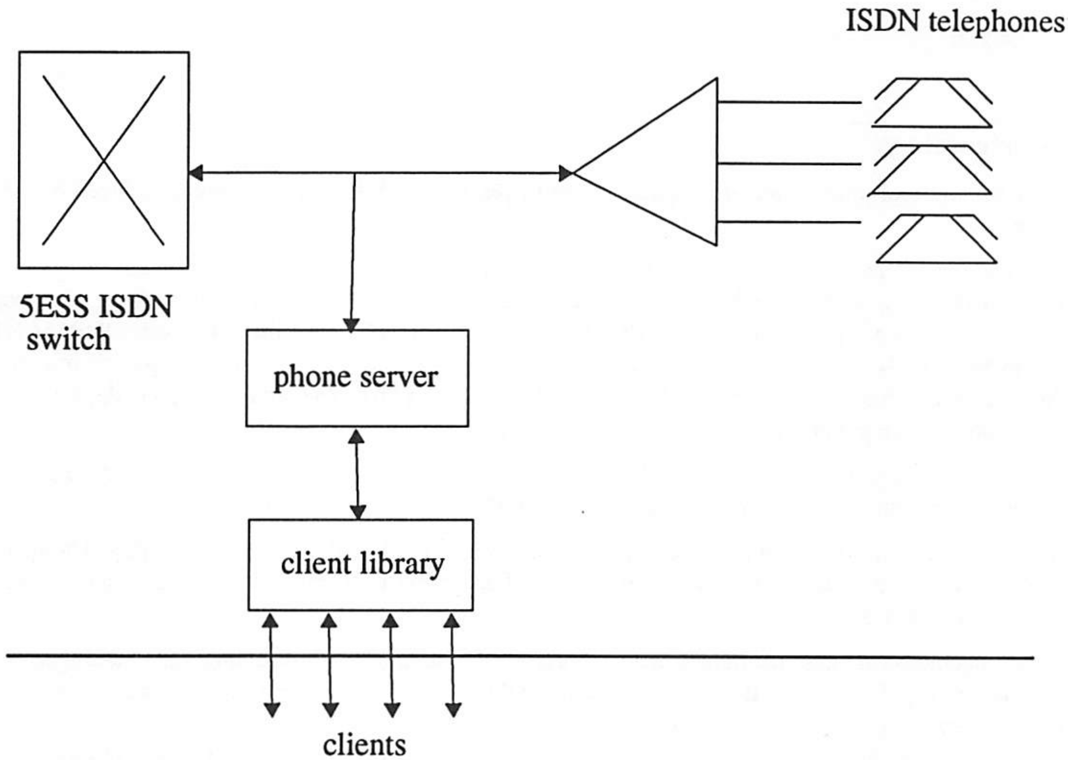


Figure 5. Architecture of the phone server

5.3 The Location Server

Another Listener, the location server, consists of a network of infra-red sensors that are placed around the building, in hallways, offices, lab areas and conference rooms. Participating members of the user community may choose to wear a badge, called the *active badge*, which will transmit and receive infra-red packets. Since each active badge has a unique code, this code and the owner of the badge can be mapped in a database.

When a user wearing his badge (designed and built by Olivetti Research Laboratory in Cambridge, England), moves around in the building, and the badge is sighted by sensors, the sensors register the unique ID of the badge and report it to the sensor server. The sensor server translates the identity of the reporting sensor to its location (e.g. East Hallway, Conference room, Lab area or Office 355, etc.) and translates the ID of the badge to its owner, thus ascertaining the location of the user (see Figure 6). The sensors are polled every 5 seconds to check badge sightings. By the time the events propagate to the activity server, there is a latency on the order of a few seconds.

6 The Activity Server

The activity server inspects the abstracted data from the Listeners and looks for overlapping activities. It filters events from the Listeners and updates its internal data structures. There is a rule-based inference engine [11] that operates on the incoming events to arrive at a high-level abstraction of the activities of the user community.

6.1 Representation

The activity server needs to keep state on a number of real world entities such as servers (Listeners), ma-

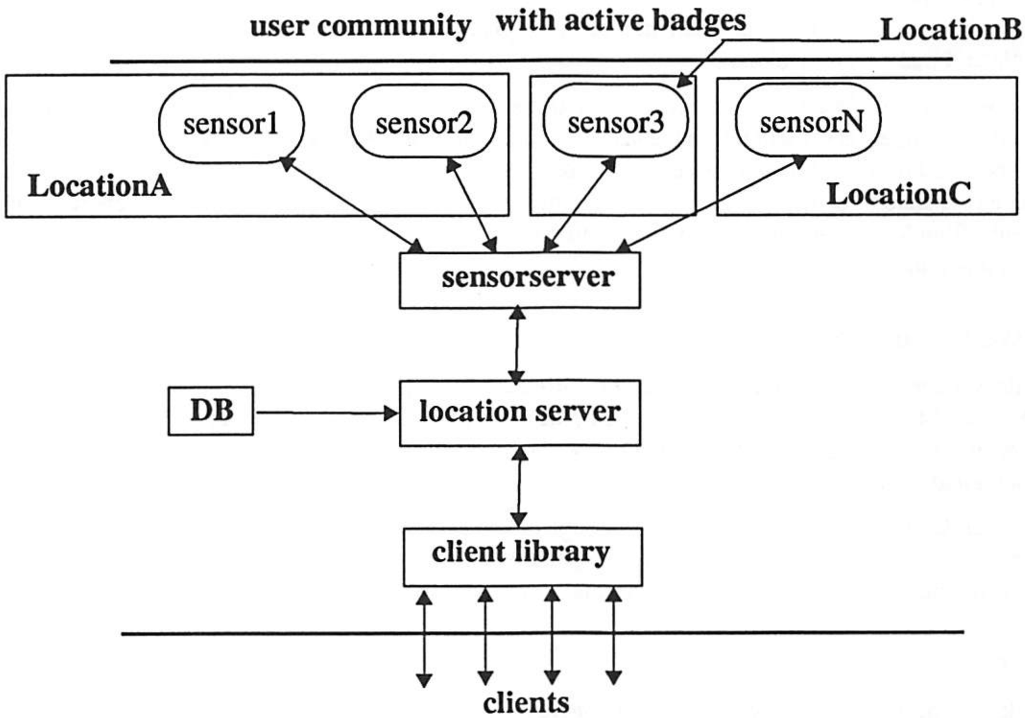


Figure 6. Architecture of the location server

chines, places and people. These entities are modeled as data objects which are described below.

6.1.1 Person object

The person object consists of a person's name which is used as an addressing token, their office location, their office phone and a timestamp. The assertions of the inference engine pertaining to this user are saved as current state and the degree of fit (given by confidence levels between 0 and 100) into the model of busy, alone or not around is saved. The head and tail pointers of the list of all significant events pertaining to the person are also saved.

6.1.2 Place object

The place object consists of the name, for addressing and informational purposes, and the update timestamp. The head and tail pointers of a list consisting of all occupants who entered (and may have left) will be maintained. Head and tail pointers of the list of all occupants currently in the room will also be maintained. By definition, users who do not have an exit timestamp are current occupants. Therefore, the list of current occupants is really a list of pointers to a subset of all occupants.

6.1.3 Server object

The server object maintains the name, the server state (up or down) and the last down time.

6.2 Mechanism

6.2.1 Startup

When the activity server starts up there are a number of initializations. It must open a socket in the advertised port to add the service; clients may then connect to the server via this port. It saves the old log file that saves system activity and creates a new one. Routine information, such the help file that will be sent to clients upon

request, and phone number, office number, usernames associated with the user community is cached. An interrupt handler is registered so that when an interrupt signal is received, the server can close sockets, update the log file and make a graceful exit.

Next, connection to all the Listeners are attempted. If the connection is successful, the initial state information is solicited. The finger server will return the least idle time and hosts of all users; the location server will return the location of all users and the phone server will return the current phone state of office phones of all users. Once the initial state information is received, each of the Listeners are given commands to report asynchronous events. Should the attempt to connect to the Listeners fail, a timer is set so that a reconnection can be attempted when the timer expires.

6.2.2 Event handling

All events, synchronous and asynchronous, pass through the event handler. The event handler is able to distinguish which Listener is reporting by inspecting the socket the information is received on; it unbundles the data accordingly and updates the data structures pertaining to that particular Listener. The event is discarded if the unbundled data reveals that the activity server does not care about that data.

The event handler timestamps all events, even though all the Listeners return their data with a timestamp. This is necessary because system times vary; a single point of timestamping on a single machine will provide a standard reference point to compare temporal information of events.

6.3 Rules

Some rules are necessary to draw higher level conclusions and to resolve conflicting data from the various Listeners or accentuate data that is in agreement across more than one Listener. Each event triggers some number of rules. Each rule has one or more conditions and one or more assertions each of which will have a confidence level associated with it. It is also possible to assign weights to some of the conditions to affect the confidence level. For instance, for someone who uses the phone rarely, the rare use of the phone may be very significant. Hence, the weights on the phone events can be increased; this will affect the decision making in building the activity model.

6.4 History Mechanism

As mentioned earlier, the person object saves all the events that affect it and the place object saves enter and exit times of all occupants. There must be a mechanism of pruning this list, however, otherwise the lists could extend beyond control. Hence, a window of interest that extends from the present time to a duration in the past is introduced. This window of interest can be set to be different for different Listeners. Events occurring beyond the window of interest are removed from the lists that form the internal state.

In addition, some rules in the inference engine can help compact the amount of state that is maintained. For instance, if a user is seen in location A and is "seen" at UNKNOWN location and is again seen at location A, the inference engine will check other Listeners and consider the second event an anomaly, i.e., the active badge was not visible to any sensor, and will discard the event and the changes it brought about.

7 Clients

There are a number of clients that rely on the activity server, and their use will test the information embedded in the activity server. Some of the clients are described below.

The *Directory Client* gives information about a particular person or a group as a whole. For example, the server can return the users' activity on the phone, the workstation or location around the building; it will also return one of the three activity classifications with its estimate of confidence in the assertion: busy, free or not around. For the busy category, the reason (on the phone, another person visiting the office, meeting in the conference room, or visiting another office) is also provided. This client generates simple text in English which can, in turn, be piped to other hardware such as a Dectalk text-to-speech synthesizer so that synthesized information can be received over analog phone lines from a remote site.

The *History Client* gives a summary of the activity within a reasonable duration of time (e.g., last hour, today, etc.) in textual format. When the user queries, this client provides breakdown of time spent on various activities.

The *Watcher Client* is a graphical interface to a client that updates the activities of participating members of the group; this client is an enhancement of a previous effort [10]. Watcher is an X Window System application that displays bitmapped images identifying other users. The display is updated whenever any user changes state on the phone, in physical location or on the network of workstations. Many tracking and messaging facilities are also available. Figure 7 shows how a lineup of users can be displayed (the users get to choose their own cartoon characters or a bitmap of their own pictures); the lower window shows how the user information may be displayed.

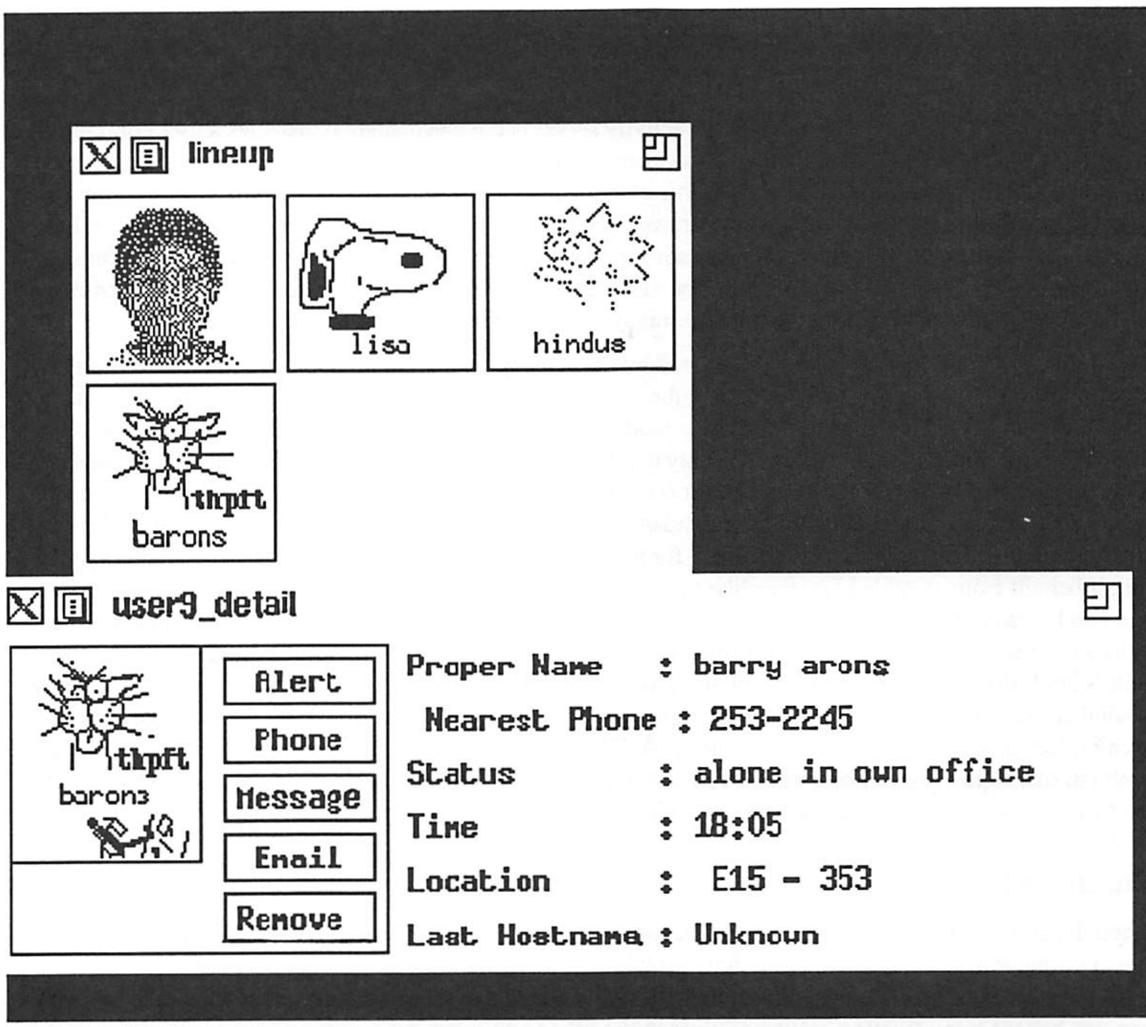


Figure 7. An example of Watcher

8 Discussion

A system like the activity server might uncover privacy issues. As emphasized earlier, a tool such as the activity server can be used very effectively in small groups of trusting individuals (e.g., project teams). However, the amount of information that the server receives can be controlled by the user himself using a customization directive alterable only by that individual. Therefore, if a user does not want others to get telephone information, for instance, the activity server will shift the emphasis to other Listeners. Having the user decide the inputs into his own model is not new; some user modeling systems have allowed the user to "edit" the model to enhance the model [5]. This "editing" concept can be used to enhance the model and also preserve user pri-

vacy. In addition, in future implementations, each of the Listeners and/or the activity server may require a password so that the information is provided only to trusted clients.

The reliability of the activity server may be undermined by many factors. The foremost factor is the varying latency in receipt of events from the Listeners. For instance, the phone server reports with minimum latency while the location server has some latency. The finger server may have variable latency depending on net traffic and the kinds of machines polled. Hence, for very fine-grained, up-to-the-second activity reports the activity server would not be suitable. Secondly, the reliability depends, to a large extent, on external factors such as whether or not the users are wearing their active badges. For instance, if a user is at a meeting in the conference room and is not wearing her active badge, the finger server is the only Listener that is useful. If the mouse on the machine that the user is logged on is moved slightly for any reason, the finger server will report an active terminal. The inference, the wrong one in actuality, would be that the user is active at that machine. Barring these constraints, the activity server can provide enough information to make reasonable activity classifications.

The principal contribution of a system like the activity server is the mechanism it provides a user community to coordinate each other's office activities. This can ease scheduling problems, reduce phone tags, and, in general, induce more productive office work. To this end, the activity server consults a number of unobtrusive Listeners, constructs an informal model and draws conclusions. Although much user modeling work relies upon a single source of information, such as user-input, eye tracker information, the activity server benefits from three different sources of information. One of the sources, the finger server, was implemented expressly with the activity server in mind, but it can have many uses independent of the activity server, as well.

There are several areas in which the work started by the activity server can progress. The first area is an online calendar, which although more static over the course of a day, is certainly very dynamic over the course of many days or weeks. It would be possible to modify the Unix calendar so that its granularity would be an hour or half hour slots rather than a day. The only drawback would be that, unlike other Listeners, the calendar would require direct involvement from the user (i.e., update the calendar regularly). Unless the user is conscientious about updating, the events that the calendar Listener can provide will be too sparse for them to be very useful in modeling the activities. Nevertheless, for projections on user activities in the future, the activity server might benefit from an added Listener like the calendar server. Secondly, the actual utility of the activity server can be gauged by a usability study. It would be interesting to find out whether or not people use the activity server actively. Investigating informed users' apprehensions or lack thereof towards a system that monitors unobtrusively might shed light on the actual utility of systems like the activity server. Thirdly, many potendial clients can benefit from the activity server. Some earlier work has concentrated on messaging services after locating the user on a local-area network [8]; the Watcher Client provides some messaging service as well but this capability can be greatly augmented. Finally, the information provided by the Listeners can be building blocks to a more rigorous user modeling.

9 Conclusion

The activity server is yet another tool that can be used in an office environment. It is unique, however, because it uses multiple information gathering techniques which provide redundant and possibly conflicting information to construct an informal model of user activities. It is hoped that the knowledge embedded in the activity server can benefit small trusting groups of users in the office environment.

10 Acknowledgements

Chi Wong wrote the majority of the phone server code and Derek Atkins along with Jim Davis wrote most the location server. Steve Tufty wrote the Watcher client. The author would like to thank Barry Arons, Debby Hindus and Mike Hawley for making constructive criticisms to earlier drafts of this paper. And finally, Chris Schmandt deserves many thanks for posing the problem and giving many helpful suggestions.

11 References

[1] Philip E. Agre. The Dynamic Structure of Everyday Life. MIT Artificial Intelligence Laboratory, Technical Report 1085, 1988.

[2] Philip E. Agre and David Chapman. What are plans for? MIT Artificial Intelligence Laboratory, AI Memo 1050, 1988.

[3] Clarence A. Ellis. Formal and informal models of office activity. Technical document. Xerox PARC, 1984.

[4] Henry Kautz. A circumscriptive theory of plan recognition. Intentions in Communications (Eds. Phillip R. Cohen, Jerry Morgan and Martha E. Pollack). MIT Press, Cambridge, Mass. 1990.

[5] Judy Kay. *um*: A toolkit for user modelling. *Second International Workshop on User Modeling*. March 30-April 1, 1990.

[6] A. Kobsa & W. Wahlster (Eds.). User Models in Dialog systems. New York. Springer Verlag. 1989.

[7] Martha E. Pollack. Plans as complex mental attitudes. Intentions in Communications (Eds. Phillip R. Cohen, Jerry Morgan and Martha E. Pollack). MIT Press, Cambridge, Mass. 1990.

[8] L.F.G. Soares, S. L. Martins, T.L.P. Bastos, N.R. Ribeiro and R.C.S. Cordeiro. LAN Based Real Time Audio-Data Systems. *Proceedings of ACM SIGOIS '90 Conference on Office Information Systems.* 1990. pp. 152-157.

[9] L. Suchman. Plans and situated actions: The problem of human-machine communication. Cambridge University Press, New York, 1987.

[10] Steven Tufty. Watcher. MIT Bachelor's Thesis. 1990.

[11] Patrick H. Winston. Artificial Intelligence. Addison-Wesley Publishing Company, Reading, MA, 1984.

Sanjay Manandhar recently received his MS from Massachusetts Institute of Technology. As a grauduate student and a research assistant with the Speech Research Group at the MIT Media Lab, he worked on incorporating a speech recognition system to the X Window System and on audio drivers and desktop audio tools. His current interests include desktop audio, storage and transport of mixed media objects and broadband networks.

Manandhar has a BS in electrical engineering from Massachusetts Institute of Technology.

Software Technology at NeXT Computer

Avadis Tevanian, Jr.
Trey Matteson
David Jaffee
Bryan Yamamoto
NeXT Computer, Inc.

We will present an overview, by way of live demonstrations, of NeXT's software (release 2.0) illustrating underlying architecture and highlighting unique aspects, including sound and music and multimedia e-mail. We will include discussion on how the multimedia environment makes use of Mach, Objective-C, Postscript and other system features.

The window server application illustrates how NeXTstep takes advantage of Mach IPC. For instance, communication between the Display PostScript Window Server and applications uses a stream-based DPS protocol implemented using Mach messaging. This interface takes advantage of out-of-line messaging where possible. The NeXTstep Pasteboard facility also uses out-of-line messaging to communicate data efficiently between applications. Font information parsed by a server from Adobe Font Metric files is shared among all applications via messaging out-of-line data. Inter-application communication is facilitated by the NeXTstep object classes Speaker and Listener. These classes provide an Objective-C cover for Mach IPC, allowing one to send messages to an object in another task the same way you send messages to a local object. The Services facility combines Speaker/Listener and the Pasteboard mechanism, allowing a user to access the facilities of one application from the another application on his system.

Another feature, The NeXT Music Kit™, is a library of objects for DSP sound synthesis, MIDI processing, and music manipulations. Although oriented around the needs of music, its functionality is quite general and includes Objective-C message sequencing and real-time digital signal processing. Applications to the fields of multi-media data visualization, animation, games, and psycho-acoustic research, as well as music, will be discussed and demonstrated.

Finally, we'll describe NeXTmail, a multi-media electronic mail application which combines the media of attributed text, sound, and graphics to facilitate interpersonal communication. We'll describe how NeXTmail makes use of its rich operating environment: the NeXT hardware, Mach, and NeXTstep, to provide integration of the various media.

MAEstro — A Distributed Multimedia Authoring Environment *

George D. Drapeau
Stanford University
Stanford, CA 94305-3090
drapeau@air.stanford.edu

Howard Greenfield
Sun Microsystems
Mountain View, CA
howardg@sun.com

Abstract

Current multimedia authoring environments are typically designed as self-contained systems; one application is completely responsible for the creation of multimedia documents. The number of media supported by such systems is usually small, and accommodating new media typically requires rewriting the authoring system.

Meanwhile, workstation vendors are adding multimedia support to distinguish their products from those of their competitors. As a result, new media are being introduced on the workstation platform so quickly that monolithic authoring systems incorporating today's state-of-the-art media are often obsolete within a year.

MAEstro was designed for extensibility. The key to **MAEstro** is its inter-application messaging system, similar to NeXT's Speaker-Listener protocol [1]. Through the messaging system, an authoring application controls a number of "media editors" (applications responsible for the manipulation of a particular medium or source of information). When creating a multimedia document, the authoring application asks the media editors for information about their current selections; during playback, the authoring application sends messages to the media editors telling them to perform media selections.

At present, **MAEstro** consists of a suite of applications on Sun and NeXT workstations. Media editors on the Sun support text, CD audio, videodiscs, and a text search engine. The NeXT runs a MIDI music editor. The authoring application is a timeline editor running on the Sun that allows authors to create multimedia documents using any combination of media on both workstations. For example, an author can create a multimedia document that synchronizes NeXT-controlled MIDI synthesizers with video controlled by the Sun.

1 Introduction

Multimedia authoring environments are typically centralized authoring environments, single stand-alone applications that provide built-in support for a number of media. Macromind's MediaMaker, Apple's Hypercard, and Imagine's MediaStation are examples of this category. A centralized authoring environment provides built-in support for a limited set of media, and the interface is designed specifically for those media. Since the authoring environment tries to support several media, support for any one medium is generally much weaker than that of an application devoted solely to that medium. For example, text support in Hypercard is not as powerful as that provided by Microsoft Word.

Centralized authoring environments provide single-source support of multiple media by trading away flexibility and power in any one of them. Although some environments provide hooks through which programmers can add support for new media, the user interface is not designed to handle them coherently. In addition, centralized authoring environments are designed explicitly with one model of authorship in mind and so are not flexible enough to accommodate new authorship models. Accommodating new media and new

*This project is part of a research effort between Stanford University, Academic Information Resources and Sun Microsystems' Collaborative Research and Worldwide Education and Research Marketing groups. We gratefully acknowledge Sun's support and funding of this project.

styles of authorship is necessary because change is the norm in the multimedia market and will remain so for the next few years.

Some have addressed these issues through the creation of courseware, customized software for a particular instructional or research domain. Courseware is typically written for a particular faculty member, the goal being to provide an authoring model specifically for that faculty's curriculum. The customized nature of courseware gives faculty the choice of media they want but does so on a case-by-case basis, thereby limiting the potential audience that can benefit from courseware.

In contrast to the courseware model, we have addressed the issue of wide access to media by creating **MAEstro**, a workstation-based multimedia authoring environment that focuses on authorship of multimedia documents. Our goal is to create a rich environment that is simple to use while providing support for a wide variety of media. **MAEstro** is scalable so that authors can create multimedia documents with whatever media are available to them. A student can create multimedia documents in public workstation labs that do not have direct access to "hard" media such as videodisc and compact disc players but may have text, graphics, and some audio capabilities.

The project is currently planning on two delivery sites within the Stanford campus: a small multimedia lab providing a wide variety of audio and video capabilities that is accessible to faculty and students at Stanford, and public workstation clusters that provide software-only media but none of the more exotic media supported by the multimedia lab. The two sites were chosen to test the scalability of **MAEstro** and to deliver the environment to a large community of potential authors.

Section 2 describes the issues that influenced the design of the **MAEstro** environment. Section 3 lists the components of the environment. Section 4 explains the authorship model underlying the design of **MAEstro**. Section 5 describes the inter-application messaging system designed to support the **MAEstro** model of authorship. Section 6 discusses plans to add new media and functionality to the environment. Section 7 discusses benefits and problems of the system.

2 Design Issues

New media are introduced to the workstation market every few months, making it extremely difficult to create a stand-alone authoring environment that coherently supports current multimedia products. Even if such an environment were built, new hardware and software introduced over the next year would likely make the authoring environment obsolete. Our greatest problem during the design process was dealing with uncertainty: which media should be supported? What should our authoring application look like? How should new media be accommodated? Who will be our authors? How do we work user input into future designs? How do we reduce the learning curve for new authors?

Section 2.1 discusses the factors that influenced the implementation of **MAEstro**. Section 2.2 defines the term "media" as used in the **MAEstro** environment. The **MAEstro** notion of authorship is defined in Section 2.3. Section 2.4 contrasts the **MAEstro** model of multimedia authorship with the more traditional courseware model.

2.1 Requirements

Demand for media has grown to a point where most people want to take advantage of multimedia capabilities in their own software offerings. However, different groups of people have different requirements of media; for example, a music researcher might be interested only in MIDI-controlled audio and simple graphics, while a graduate student in German Studies might want to synchronize video with digitized audio in several languages; an engineering student might want to combine the visual results of a simulation engine with a textual description of the simulation. There is no one set of media that will satisfy everyone; furthermore, some require media not yet available to us. In short, we do not know which media authors will want nor how they will combine media.

We learned from the courseware model that authorship takes many forms; in essence, courseware is all about writing a new authorship model for each faculty member. This means that an authoring environment should be designed to accommodate different styles of authorship, or different ways to structure information. To impose a single style of authorship and expect it to suit everyone's purposes is a mistake.

The workstation platform is still very much in an experimental stage of multimedia development; multimedia itself is a moving target and we expect it will remain so for the next few years. To serve the computing community well, an authoring environment should be designed for extensibility so that new media can be integrated with the existing environment as they become available. At the same time, we recognize that a limited development staff affects the ability to write tools for new media; the environment should therefore allow new media to be added without perturbing any existing code if possible. A limited staff necessitates localized changes to the environment.

The primary issues driving the current implementation of **MAEstro** are the ability to allow the author to pick and choose her set of media, the ability to accommodate diverse styles of authorship, and extensibility that is as painless as possible to application programmers. Portability is also desirable since our public workstation rooms include workstations from several vendors.

The final requirement is defined by the tools available in our Sight and Sound lab, a public access room providing video and audio hardware and software for use by all students and faculty on campus. The lab dedicates machines to particular media (for example, a NeXT computer serves as an audio workstation, while a Sun controls videodisc devices). To allow authors to create multimedia documents that include both types of media, it is desirable to create an environment that allows authorship of documents that span machines; in other words, interoperability. This requirement is different than that of portability, which allows us to take an existing environment to a different computer. Interoperability allows us to use services from multiple, heterogeneous computers simultaneously.

2.2 Defining "Multimedia"

Discussion of multimedia typically includes the media of text, images, audio, and video. Some choose to categorize these media in more specific terms; for example, material describing the Intel DVI specification treats video as three components: live motion video, video special effects, and video stills [2, 3]. Our feeling is that even these forms of media cannot necessarily be classified in such broad terms, since different people will use the same basic media in different ways. For example, a musician might use three different forms of audio: digitally sampled audio that is stored on disk, audio generated by a digital signal processor (DSP), and MIDI signal data to control external synthesizers. Software to control these three sources of audio would most likely treat them differently from one another, and yet they are still generally considered "audio". This distinction drives our definition of media.

For the **MAEstro** environment, we define media as *sources of information*. Any source of information is its own medium, and usually has an application devoted to the manipulation of that source. Another way of saying this is that *media are defined by the applications written for them*. Thus, the application responsible for controlling MIDI-based audio defines MIDI as its own medium; the digital sampled audio application defines digital audio as yet another medium.

In considering the needs of humanities faculty and students we found that although they want alternate media as part of their computing environment, these media would not be the whole of that environment. They want to integrate new media with data manipulated by their own domain-specific applications. By considering any application or source of information as a medium, the scope of the term "multimedia" widens to include not only audio and video, but also text search and retrieval engines, simulation applications, visualization applications, spreadsheets, and so on. This model offers authors a richer combination of media than a more traditional definition of media. For example, we envision a researcher using text search queries and their results as part of a multimedia document then adding the researcher's own voice annotations and text from a word processor to explain the meaning of the search. The number of media possible is limited only to the number of applications available on the workstation.

2.3 Defining "Authorship"

Authorship is the process of creating a document. The nature of the document and process by which a document is created is highly variable, but the goal of the authorship process remains to create a multimedia document.

Authorship may take any number of forms. Hypercard is an authoring application in which information is structured as a series of cards the author can link together. Macromind's Director for the Macintosh is

another authoring application, revolving around the presentation of multimedia "slide shows". A musical score editor would be another authoring application; the structure of such an application's documents is obvious to musicians. The NeXT Mail application might also be considered an authoring application. The structure of a document in NeXT mail is a series of mail messages that can be plain text or "packages" that include "attachments" (documents from other applications). Authoring a document with the NeXT Mail application means composing a package that is sent to another person.

2.4 The MAEstro Model of Authorship vs. the Courseware Model

MAEstro was designed to address some of the problems inherent in the courseware model of multimedia software development. The courseware model focuses on the structured presentation of media; usually, a student reacts to an existing document by choosing from a limited number of predefined options. The **MAEstro** model of authorship focuses on the creation of documents rather than presentation of pre-formed documents.

Stanford currently has a small group of programmers who create courseware for faculty on campus. The courseware model at Stanford typically works as follows: a faculty member decides that she wants to use the computer as part of the class curriculum. The courseware development group devotes one of the group's programmers to the faculty member for some period of time. The programmer and faculty member meet initially to discuss goals and requirements. During development, the programmer meets occasionally with the faculty member to insure the accuracy of the material written into the courseware and to get feedback from the faculty member. Development time for courseware projects varies widely; projects have lasted as little as a few months and as long as three years.

The courseware development group at Stanford has produced some software of great use to faculty, and we believe the courseware development model to be a useful form of software development. However, there are a number of problems with the courseware model:

- **Development is labor-intensive.** Writing courseware is difficult and time-consuming; as a result, faculty do very little if any development of their own. The work is almost completely done by the programmer. Lending a programmer to a faculty member for some period of time means that only a limited number of documents (courseware offerings) can be created in a given time frame. The courseware development group at Stanford has seven programmers; even if each programmer could produce two complete courseware offerings per year, the vast majority of the faculty community would be neglected.

- **Productivity/success is unpredictable.** It is difficult to predict how long a courseware project will take to complete and how useful it will be once completed. The courseware development group has spent three years developing courseware that is used by a single class for a period of perhaps two weeks. The group has also spent less than a year to produce courseware that can be used by several foreign language classes for the duration of an academic term or more. Taking on a courseware development project entails a great deal of risk.

- **Courseware is often "read-only".** For most courseware, information flows from the teacher to the student; the student is simply reacting to information structured by the teacher and programmer. In many courseware offerings, the student is interacting with a pre-made, structured document that offers yes or no choices along the way. More sophisticated courseware allows the student to set a larger number of parameters; even so, the student is still not modifying or adding to the knowledge contained in the courseware. The student cannot draw out more information from the courseware than has already been programmed in. This denies the student the learning associated with the act of creation, as happens when writing a paper. Often the programmer learns more about the subject matter by creating the courseware than the student does by using it.

MAEstro addresses the problems above by taking the approach that students are authors. The job of our programmers, then, is to provide tools and applications that make authorship simpler for students. The objective of courseware is often to automate the teaching process; by making students into their own authors, we can forget about trying to automate teaching and focus on simpler tools to help students write multimedia

"papers". This model puts more burden on the students, but our hope is that the multimedia tools will engage students enough that they will be motivated to create. We believe that focusing on students creating their own multimedia documents can potentially serve a greater segment of the computing community than the courseware model.

The resulting architecture is described in the next section.

3 System Overview

MAEstro consists of four logical components:

1. Media editors
2. An authoring application
3. An inter-application messaging system
4. The Port Manager application

Just as work in an office environment is spread among specialists, authorship in **MAEstro** is spread among applications well suited to a particular medium, called *media editors*. An authoring application serves as the "office manager", coordinating the actions of the other applications; it does so by sending messages to the applications, telling them to select pieces of a document and perform them. The authoring application communicates with the other applications via an inter-application messaging system designed for the environment. Applications advertise their services by registering themselves with an application called the Port Manager. An application can query the Port Manager to discover which other applications are currently advertising their services.

Section 3.1 defines the notion of a media editor and lists media editors in the current environment. The authoring application is described in Section 3.2. The inter-application messaging system is described in Section 3.3. Section 3.4 describes the Port Manager.

3.1 Media Editors

Recall from Section 2.2 that media are defined by the applications written for them. In **MAEstro**, these applications are called "media editors". As new media are introduced to the workstation platform, media editors are written to give authors access to and control of those media. Media editors do not generally stand alone, rather they are used to create pieces of a multimedia document. The overall structure of a complete multimedia document is managed by the authoring application.

Our current environment includes the following media editors:

- **cdEdit**— An application used by authors to annotate audio compact discs. The application stores a list of start and stop points corresponding to audio segments on the disc.
- **videoEdit**— An application for annotating videodiscs. The author creates lists of start and end points that define single frames of video or complete scenes.
- **QuoteMaker**— An application for quoting existing text, as opposed to a full word processing application. The author opens a text for browsing; selecting part of the text creates a "quote" that can be saved as part of a list of quotes. The author later displays the quote in large type on a separate window as part of a multimedia presentation.
- **Searcher**— A text search and retrieval engine application used primarily by humanities researchers as an analysis tool for online text. The application stores a list of queries the author makes of a particular text; the author can later open the list of queries and resume searching on that text.

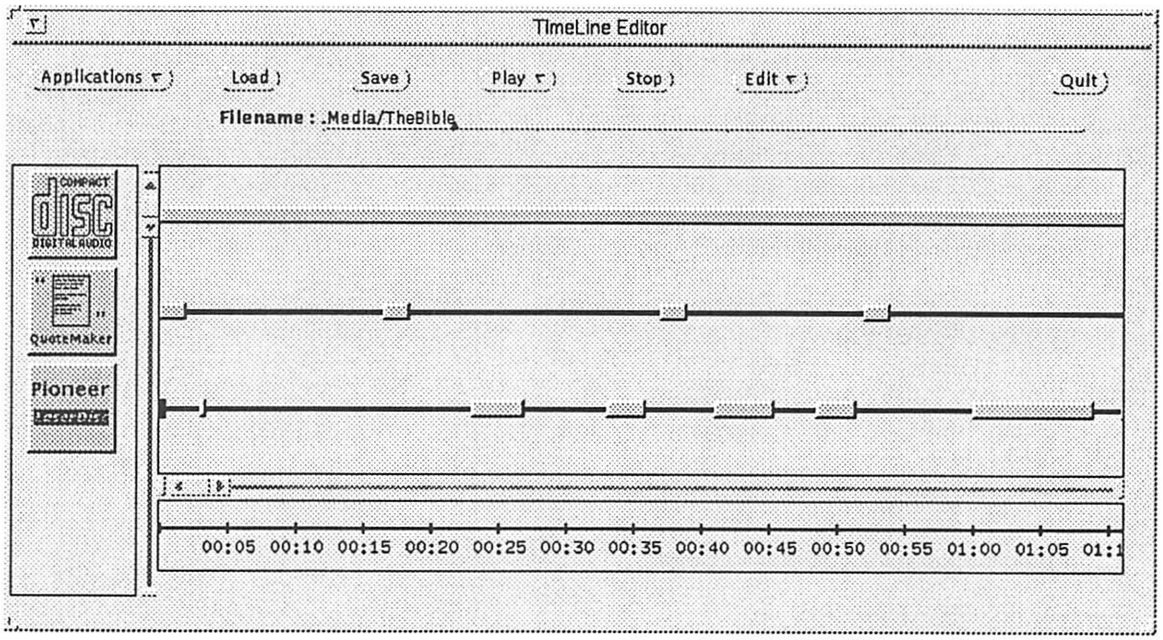


Figure 1: *A TimeLine Editor document coordinating three media editors: cdEdit, QuoteMaker, and videoEdit.*

3.2 The Authoring Application

An authoring application defines the structure of a multimedia document. It provides a paradigm of authorship by offering metaphors and structures with which the author is familiar. The authoring application supports a particular model of authorship just as a media editor supports a particular source of information.

We have chosen to implement a timeline paradigm of authorship for our first authoring application. The TimeLine Editor application represents documents as a number of "tracks" of time, one track for each medium in the document (see Figure 1). It is a simple model that works well for linear presentations requiring no interaction on the viewer's part. Using a timeline model allowed us to concentrate on authorship of documents (as opposed to interacting with already existing documents).

The authoring application does not directly control media; instead, it controls the actions of the media editors which do the actual media manipulation. The authoring application exerts this control via the inter-application messaging system. The interaction between the authoring application and the media editors is explained in Section 4.

3.3 The Inter-Application Messaging System

At the heart of **MAEstro** is an inter-application messaging system similar to the Speaker-Listener protocol used on the NeXT computer. Unlike Speaker-Listener, the **MAEstro** messaging system is currently implemented with Sun RPC's [4], making the **MAEstro** messaging system available on a wide variety of systems.

Each application in the **MAEstro** environment (media editors and the authoring application) uses the inter-application messaging system for communication with other applications. Messages are provided for applications to advertise their services, to request state information from other applications, and to request other applications to open documents and perform media segments within documents. A typical use of the protocol is for an authoring application to request a media editor to open a document, select part of that document, and perform that selection. The messages are few and simple, designed to transmit as little data among applications as possible.

Note that the **MAEstro** protocol does not address the problems of synchronization of media and guaranteed network delivery of continuous media [8, 9]. Our programmers follow specific guidelines in using the protocol to help insure synchronization, but trying to take into account the trade-offs between varying speeds of media hardware and real-time performance requirements is outside the scope of **MAEstro** in its current state.

The semantics of the **MAEstro** protocol are described more fully in Section 5.

3.4 The Port Manager

The Port Manager application is similar to the Sun "portmapper" program in function — it listens on a well-known TCP/IP protocol port for messages from applications that wish to advertise their services, and keeps an internal list of the TCP/IP port numbers passed in by the registering applications. The Port Manager serves as a rendezvous point for applications that wish to communicate with each other. The main difference with the Sun portmapper is that the Port Manager associates TCP/IP port numbers with program names, whereas the Sun portmapper associates TCP/IP port numbers with RPC tuples. Associating port numbers with program names allows multiple applications to use the same RPC protocol.

The Sun portmapper program was written with the traditional client-server model of computing in mind, meaning that only one application is set up to respond to messages sent using a particular RPC protocol (Sun RPC supports message broadcasts where any number of programs may respond, but the assumption is that all responding programs are clones of the same server). The portmapper associates RPC tuples of the form *<program number, version number, procedure number>* with TCP/IP protocol port numbers; the implicit assumption is that only one application will respond to any given RPC. This discourages sharing of RPC's among peer applications that provide similar services.

In the **MAEstro** environment, all applications can provide similar services and therefore speak the same protocol; any application can be both a "client" and a "server". The **MAEstro** Port Manager associates port numbers with program names instead of with RPC tuples, since all applications speak the same protocol. In the Sun client-server model, when a server registers itself with the Sun portmapper, the server tells the portmapper the port number and RPC numbers on which it is listening [6]. In the **MAEstro** model, when an application launches it registers itself with the Port Manager, indicating the name of the application (e.g., the compact disc editor registers itself as "cdEdit"), the hostname on which the application is running, and the port number on which it is listening for incoming messages.

Since all applications in the **MAEstro** use the same set of RPC's, applications can easily communicate with new services (applications) added at any time, since the new services use a protocol already known to the other applications. Applications can become aware of new services by querying the Port Manager.

4 Model of Authorship

To create a multimedia document, an author uses one application for each medium to be included in the final document plus the authoring application used to structure the media. The data itself stays in the media editors, which send representations of their data to the authoring application. In other words, the authoring application does not directly store media but instead stores pointers to media. The Sun Link Service [5] follows this paradigm of linking documents.

Central to the **MAEstro** model of authorship is the notion of a *media segment*. A media segment is part of a document managed by a media editor. For a text editor, a media segment would be a selection of text that might be anything from a single character to a chapter in a book or more; for a videodisc application a media segment would be expressed in terms of frames of video.

Consider the following scenario as an illustration of the model of authorship we envision for **MAEstro**: A music student is to analyze a Beethoven symphony as a term project. The student has collected material for the project from several sources — a performance of the symphony available on compact disc, a videodisc with a performance of the work by a major symphony orchestra, and historical material from several textbooks.

The student inserts the CD into the workstation's CD player and opens the cdEdit media editor. The student uses cdEdit to listen to the performance, stopping along the way to mark passages that she might wish to use as musical "quotes" in the final paper. The student may listen to the disc several times, adding passages to the edit list. It is important to note that at this point the student is collecting notes to be included in a rough draft, so the quality of the edits need not be perfect; rough starting and ending points will do. When finished, the student saves the cdEdit document and moves on to the video material.

The student now launches videoEdit to watch the videodisc performance of the work. The student uses videoEdit in the same way she used cdEdit; she watches the performance, stopping occasionally to add

interesting segments to an edit list. The student finishes making a rough edit list and saves the videoEdit document, then launches the word processor to write the textual portion of the paper. When a rough draft is finished, the student saves the document.

At this point in the authoring session, the component media are roughly organized but have not yet been combined into any overall structure. The student now launches the TimeLine Editor to provide that structure.

The student selects an edit from cdEdit, then clicks on the appropriate track in the TimeLine Editor. A bar appears representing the cdEdit segment's duration. The student places media segments on different places in the timeline, selecting a media segment from one of the editors then clicking on the place in the timeline where the segment is to be performed. When a timeline document has been laid out, the student presses the "Play" button to observe the presentation. The TimeLine Editor sends messages to the media editors at the appropriate times, telling them to perform their segments.

The student observes the timeline she just created and finds that a particular segment of video was a little too long. She goes back to videoEdit and plays the segment, then changes the start and end points until she is satisfied with the result. She re-saves the videoEdit document, goes back to the TimeLine Editor, and replays the document to observe the effect of the modified video segment.

The process of refinement continues in this fashion — the student uses the TimeLine Editor to perform the document, then uses the media editors to do fine tuning of individual media segments, then again to the TimeLine Editor to observe the changes until she is satisfied with the presentation.

5 The MAEstro Messaging System

New media may use as yet unknown data formats, making it difficult if not impossible for an authoring system to support every medium's data format. Thus, the **MAEstro** protocol was designed as a "remote control" protocol instead of a data transfer protocol. By "data transfer protocol" we mean a protocol designed for the transmission of digital media from one application to another. By "remote control protocol" we mean a protocol used to send commands from one application to another. Using a data transfer protocol, an application would transfer digital audio data to a remote application by sending the data directly to the remote application. For two applications to transfer digital audio data in the **MAEstro** environment, an application sends a message asking the remote application to open the document containing the audio data instead of sending the audio data directly. The difference in protocols is analogous to copying memory from place to place (the data transfer protocol) versus copying a pointer to the memory (the remote control protocol).

An individual media segment is represented by:

- Application name: The name of the application (media editor) that created the segment.
- Document name: The name of the document containing the segment.
- Segment information: A representation of the segment itself. The **MAEstro** term for this is a *selection*.
- Duration information: An estimate of the time necessary to perform the segment.

Applications in the **MAEstro** environment define their own notions of "document", "selection", and "performance". However, the notion of time taken to perform a selection has a concrete interpretation shared by all **MAEstro** applications, and is measured in milliseconds. The flexibility and power of the **MAEstro** protocol lies in the ability of each application to define its own notions of document, selection, and performance.

Section 5.1 describes the `Selection` data structure used to represent **MAEstro** selections. Section 5.2 discusses the set of messages sent among applications in the environment. Section 5.3 gives some examples showing how different applications might interpret the notions of document, selection, and performance. Section 5.4 shows how a multimedia document can be constructed from media editors on several hosts simultaneously.

5.1 Selections

The authoring application does not concern itself with specific data formats used by each media editor; instead, it asks media editors for representations of their data. These representations are stored in a data structure called a `Selection`, which looks like this:

```
struct Selection
{
  int start;
  int end;
  int duration;
};
```

An application programmer decides how the application will encode media segments into the two fields `start` and `end`. When the authoring application asks a media editor to perform a selection, the media editor must be able to decode the `start` and `end` fields to identify the original media segment.

The `duration` field serves as an estimate of the time needed to perform a selection, and is specified in milliseconds. This is the only field of the `Selection` structure that is interpreted by the authoring application.

5.2 Messages

There are two types of messages used by **MAEstro** applications: messages sent to other applications and messages sent to the Port Manager. Messages sent to other applications are used for opening documents and performing media segments. Messages sent to the Port Manager are used to register services or to ask for information about application services currently being offered.

During authorship of a document, the authoring application asks a media editor for the name of its currently open document and the current selection within that document. **MAEstro** supplies two messages, `GetCurrentDocName` and `GetSelection`, for this purpose. During performance of a document the authoring application asks a media editor to open a particular document, to select part of the document, then to perform the selection. The messages `OpenDocument`, `SetSelection`, and `PerformSelection` are provided for this purpose. Any application may send these messages to any other application, but at present only the authoring application sends these messages.

An application advertises its services by registering itself with the Port Manager. When an application is about to quit or no longer wishes to advertise its services, it must "unregister" itself with the Port Manager. The protocol provides the messages `ConnectWithPortMgr` and `DisconnectFromPortMgr` for these two actions.

The authoring application needs to be aware of applications currently advertising their services. To obtain this information the message `GetOpenApps` is sent to the Port Manager; the Port Manager returns a list of applications currently advertising their services. Any application may send this message to the Port Manager, but typically the authoring application is the only one concerned with which services are currently available. To ask for a specific service by name, an application sends the message `GetPortFromName` to the Port Manager.

5.3 Interpreting Messages

Media editors interpret the notions of document, selection, and performance as best fit their particular media. This section describes how three applications (a text editor, cdEdit, and the Searcher) interpret these three notions.

5.3.1 Text Editor

A document for a text editor is the paper, memo, letter, etc., typed in by the author. When the text editor receives an `OpenDocument` message from another application, the text editor interprets the string passed to it as the name of a file to open. If the file does not exist, the text editor returns an error code.

Selections are represented as the starting and ending bytes of a sequence of text within the current document. When the text editor receives a `SetSelection` message, the text editor tries to highlight the range of bytes specified by the incoming message. If there is no current document or the given range is invalid, the text editor returns an error code.

The notion of the time necessary to perform a selection is less clear. A text editor does not normally associate time with the text in a document. However, since the protocol requires an application to return an estimate of how long the current selection will take to "perform", the text editor has two choices: return a hard-coded value of perhaps zero milliseconds or one second, or use a heuristic to estimate performance time. The heuristic might be: "based on an average reading speed of 250 words per minute and the length of the current selection, calculate the number of seconds to read the current selection." The **MAEstro** text editor reports a hard-coded value to the authoring application and displays the selected text in another window (in large type), but does not remove the display window when the time has elapsed. A future version of the text editor will provide the author a choice between a heuristically-derived duration and the author's own duration.

5.3.2 cdEdit

The cdEdit application stores documents as edit lists. In addition to start and end points, cdEdit stores a brief text label to describe each edit.

The cdEdit application uses only the `start` field of the Selection structure, ignoring the `end` field. The `start` field is used as an index into the edit list. The `duration` field represents the time to play the edit.

Performance for cdEdit is clear; performing a selection means playing from the start of an edit to its end.

5.3.3 Searcher

Searcher documents store the name of the text being searched and a list of queries the author applies to that text. When a Searcher document is opened, the requested text is opened and the author can redo any of the previously saved queries or she can generate new queries.

A selection is a pointer to one of the queries in a document and a pointer to one of the results, or "matches", from that query. The Searcher uses the `start` field of a `Selection` as an index into the list of previously saved queries, and the `end` field as a pointer to one of the results generated by the query. For example, a selection's `start` and `end` values of 3 and 14 would be interpreted as the third query of the current document and the fourteenth match generated by that query.

The Searcher performs a selection by sending a query and displaying the results. To perform the selection indicated above, the search engine would execute the third query of the current document then display the fourteenth result of the query.

5.4 Network Control of Media

Authorship of multimedia documents is not limited to a single host; the authoring application can coordinate media editors on multiple hosts. When an application registers itself with the Port Manager, it uses the following structure, called a `Port`:

```
struct Port
{
  char* hostName;
  char* appName;
  int   portNumber;
};
```

The `hostName` field indicates the name of the host on which the application is running. The `appName` is the name under which the application advertises its services. The `portNumber` is the TCP/IP port number on which the application is listening.

When the authoring application sends the `GetOpenApps` message to the Port Manager, the Port Manager returns a list of `Ports`, one `Port` for each application that is currently listening for messages. An application

can register itself with any Port Manager on the network, and an application can ask any Port Manager which applications are registered with that Port Manager.

Viewing a multimedia document elsewhere on the network is a potential problem since some media cannot be performed over the network. For example, we cannot play CD audio over our networks although text can easily be displayed over the network with the appropriate window system. To address this problem, the authoring application keeps track of the "authorship host", the host on which a document was originally created. If the authorship host is different than the host from which the document is being played, the authoring application allows the author to specify which host(s) should be used for playback of each media editor in the document.

Perhaps the greatest disadvantage of this type of network control of media is that the network is not completely transparent. Authors should not be forced to think about the network, but considerations such as the location of files and media hardware connected to the local workstation affect transparency. While it is fortunate that the author can specify authorship and playback hosts, it is unfortunate that she must sometimes do so. Of course, if we assume a standard hardware and software configuration for delivery, the problem of network opacity diminishes since all services would be available on all hosts. We can do this to some extent, but some media (such as video) are still expensive, so not all of our workstations will have all media capabilities.

6 Future Directions

MAEstro is on a development schedule that will see its first delivery to the Stanford computing community in October. First delivery of **MAEstro** will take place at two sites: our multimedia lab will provide a variety of video and audio media, and our public workstation rooms will provide baseline services accessible to all. The authoring environment in the public areas will not be as rich as for the multimedia lab, but students will still have access to software-only media (text, search engine data, images, etc.).

Section 6.1 describes the applications currently being developed for the environment. Section 6.2 discusses our goals for ease of use and how we intend to obtain those goals. Section 6.3 discusses plans to extend the environment.

6.1 Future Applications

Although we support some form of text, audio, and video, the environment as defined by the applications above are mostly read-only in nature. Our belief is that while CD audio and videodiscs have value, the availability of writable media will afford a much wider community of authors. In keeping with this philosophy, we are developing the following applications:

- **vcrEdit**, an application to control the new NEC PC-VCR videotape recorder that accepts VHS tapes recorded on any consumer VCR.
- **DigitalTapeRecorder**, an application to record and playback sampled audio on Sparcstations.
- **ShellEdit**, an application that allows the author to spawn Unix system processes, shell scripts, etc. This is a bridge between applications that speak the **MAEstro** protocol and those that do not.

In addition to these applications, we have plans to write a NeXT-to-**MAEstro** protocol converter. The protocol converter will speak both the **MAEstro** and Speaker-Listener protocols, allowing NeXT applications to be used as **MAEstro** media editors.

We are also seeking co-developers to add media editors and alternative authoring applications in order to provide access to a wider variety of media and several paradigms of authorship from which potential authors can choose.

6.2 Ease Of Use

We are working in several ways to make **MAEstro** a simple environment to use. First, we will continue to enhance the abilities of our existing media editors, providing richer functionality to authors. Distributing

authorship among applications can work, but only if the components themselves are fully functional. Second, we will develop additional media editors as new media become available and as demand for other media becomes apparent. Third, we are trying to leverage applications with which authors are already familiar in order to reduce the learning curve. This means adding the **MAEstro** messaging system to existing applications when possible.

6.3 Extending the MAEstro Messaging System

Preliminary use of the environment has pointed out a number of messages that should be added to the protocol. Included are messages to control performance as it is happening (e.g., pausing and resuming a performance, and performing part of a selection), and messages to help overcome "window overpopulation", a cluttering of the screen that increases as the number of open applications increases. Such messages might include requests for an application to hide itself from the screen, to show itself, and to bring itself to the front of the window stack.

The current selection paradigm does not allow for overlapping selections; one selection for a particular medium must end before the next begins. One solution we are considering is additional dialog between a media editor and the authoring application about whether the media editor has the ability to start new selections while it is currently playing a selection.

The NeXT Speaker-Listener protocol is written in Objective-C and allows programmers to add messages by subclassing the two messaging objects. **MAEstro** does not yet have this flexibility, since its messaging system is built on top of Sun RPC which is written in C and is not subclassable. We hope to find a subclassable messaging mechanism to replace the current Sun RPC model. This would allow application programmers to add their own messages to be shared among a select group of applications, and would allow us to experiment with some of the additional protocol items discussed here without breaking the current environment.

7 Conclusions

MAEstro is still in the development stage; we plan to deliver the first version of the environment in October 1991. During the summer we will continue testing with a small number of authors and cycle feedback into the development process.

Section 7.1 discusses some of the benefits of the **MAEstro** model not already mentioned. Section 7.2 discusses some of the problems with **MAEstro**.

7.1 Benefits

The distributed nature of the environment allows third party vendors to enrich the authoring environment by providing full-featured applications. The environment is only as good as its media editors; therefore, the better that applications behave, the better the quality of resulting multimedia documents.

The **MAEstro** model encourages authors to use familiar tools when creating documents. By contrast, centralized authoring environments require the author to use the tools provided by the authoring application to manipulate media. For example, to edit text in Hypercard an author must use the text editing provided by Hypercard (plain text can be imported from the system clipboard, but formatted text from Microsoft Word cannot be imported). This increases the learning curve for the author, since she must learn a new text editor for each authoring application she uses. In **MAEstro**, the author can use the word processor directly in the multimedia document.

MAEstro allows for rapid prototyping of multimedia systems. Multimedia support on workstations is still largely in an experimental stage, making it difficult to predict which media types will prevail in the next few years. The ability to plug in new media without disrupting existing functionality allows **MAEstro** to track the moving target that is the workstation multimedia market.

7.2 Problems

MAEstro does not address well the problem of synchronizing media. There are no synchronization primitives in the protocol, and the environment is currently at the mercy of slow media such as videotape. Macromind's

MediaMaker addresses synchronization by supporting only media that can be interrupted at any time; a text search engine or database tool would not be allowed in the MediaMaker environment because a query cannot be interrupted in the same way that digital audio can be stopped. The result is that real-time presentation is not at all guaranteed; in fact, it is not even a primary goal of MediaMaker. We are trying to ameliorate synchronization problems by enforcing a set of guidelines when developing applications and by adding a preview feature to the TimeLine Editor that would compare a performance's actual behavior to its expected behavior and try to correct for the differences.

The environment does not support version control of documents, as does the Sun Link Service and Apple's Publish-Subscribe model of inter-application communication [7]. There is no way to tell the authoring application to update its link to a media editor's document; if the media editor's document has been changed or deleted, the author must make a new link from the authoring application herself.

Multimedia documents in **MAEstro** tend to have too many components, thus complicating delivery. Recalling the authoring scenario from Section 4, there were at least four documents involved (text, cdEdit, videoEdit, and authoring application document). Delivery of such a document would be problematic in any system since "hard" media are involved (a CD and videodisc would have to be delivered with the documents), but packaging a number of files for delivery has different implications than sending a single file regardless of whether hard media are involved. Part of this problem is handled by network control of media, but there are some media that are not yet amenable to network access; the media themselves must be at the viewer's desk, and it is these media that create the delivery problem. Perhaps the **MAEstro** model joined with applications capable of delivering continuous digital media will alleviate the delivery problem.

The authoring environment in its current state suffers from window overpopulation. Possible solutions are to increase screen real estate (but to what extent? 6000 by 4000 pixels? A wall-sized display?), or add window manager-like messages to the protocol that allow the authoring application to tell applications to hide themselves when not being used, to show themselves when needed, and to hide unnecessary windows during performance of a document.

We have not yet addressed security issues raised by the ability to use media editors over the network. We do not see security as a serious problem in the short term (the X Window System as delivered by the X Consortium is an example of an environment that has so far survived with a limited security scheme), but this by no means diminishes the serious nature of the problem.

8 Acknowledgments

Many thanks go to Teck-Joo Chua for the excellent work he has done implementing the TimeLine Editor and the first versions of most of the other applications in the current environment, as well as his sound advice on current development issues. We would also like to thank Bryan Clair for his help in the early design and implementation phases of the project. Thanks go to the development team of Derek Lai, Wee-Lee Lim, Bryant Marks, and Mark Warren for their work. Additional thanks to David Finkelstein, Yaron Gold, John Interrante, and Johnson Lam for their valuable advice. Special thanks go to Steve Loving for his ideas of student authorship that inspired the project.

References

[1] *NeXT System Reference Manual—Release 1.0 Edition*, NeXT Computer, Redwood City, CA.

[2] Kevin Harney, Mike Keith, Gary Lavelle, Lawrence D. Ryan, Daniel J. Stark, "The i750 Video Processor: A Total Multimedia Solution," *Communications of the ACM*, 64-78, April, 1991.

[3] G. David Ripley, "DVI — A Digital Multimedia Technology," *Communications of the ACM*, 811-822, July, 1989.

[4] *Network Programming Guide*, Sun Microsystems, Inc., Mountain View, CA.

[5] Amy Pearl, "Sun's Link Service: A Protocol for Open Linking," *Proceedings of ACM Hypertext '89*, November, 1989.

[6] W. Richard Stevens, *Unix Network Programming*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[7] J. Paul Grayson, Chris Espinosa, Martin Dunsmuir, Mike Edwards, Bud Tribble, "Operating Systems and Graphic User Interfaces," *SIGGRAPH '89 Panel Proceedings*, December 1989.

[8] Domenico Ferrari, "Guaranteeing Performance for Real-Time Communication in Wide-Area Networks," U.C. Berkeley Technical Report UCB/CSD 88/485, December, 1988.

[9] Domenico Ferrari, "Real-Time Communication in Packet-Switching Wide-Area Networks," International Computer Science Institute Technical Report TR-89-022, May, 1989.

George Drapeau is Workstation Environments Specialist in the Academic Information Resources (AIR) Department at Stanford University. He received the B.S. in Computer Science (1986) and M.S. in Computer Science (1988), both at the University of Southern California. His current interests are multimedia computing, user interfaces, and window systems.

Howard Greenfield did his graduate work in Interactive Educational Technology at Stanford University. His work prior to coming to Sun, included computer consulting to diverse business and educational organizations, software development for Computer Science Press, as well as instruction. He also has worked as a researcher for the Apple Classroom of Tomorrow (ACOT) group. He is currently managing an Interactive Media Systems group at Sun specializing in systems integration and deployment of hypertext and multimedia technologies.

Newspace: Mass Media and Personal Computing

Walter Bender, Håkon Lie, Jonathan Orwant, Laura Teodosio, Nathan Abramson
Electronic Publishing Group, MIT Media Lab, 20 Ames St. Cambridge, MA 02139
voice:(617)253-7331 fax:(617)258-6264
walter@media-lab.media.mit.edu

Abstract: This study describes the implementation of a display application, Newspace, that offers a broadsheet-sized electronic news presentation to the reader. The use of "paper-quality" displays coupled with personal computing gives us a range of new possibilities in content selection, imagery, typography, and human interaction. A computational intermediary, acting in concert and on behalf of the "reader" makes possible new publishing styles and forms. Newspace invites instant updates and spontaneous interaction, while retaining the notion of the edition, with a "front page," headlines, simultaneous presentations and juxtapositions.

The application is not limited to traditional news retrieval. The news is augmented with information from local and remote databases, including electronic mail. This compilation of data is presented in the style of news media, complementing rather than supplanting it.

The combination and juxtaposition of multiple forms of media is being explored. Salient features of news stories, such as the non-lexical cues in audio and video news, are abstracted and stored in a database. The information is used in augmentation and linking to stories in remote databases, to translate information across media types, and to display the news in styles amenable to individual users. Further personalization is accomplished by maintaining dynamic user models. These models include information such as the user's schedule and topics of interest. The system monitors user actions to bolster both precision and accuracy of the model over time.

1.0 Introduction

Three emerging technologies are altering the way we think about the relationship between mass media and the individual: (1) the plethora of digital communication channels, (2) the ubiquity of personal computing, and (3) advances in display technology. In considering all three of these advances together, there is an opportunity to reexamine the current status quo of both mass media and personal computing.

This paper describes the implementation of an application that offers a broadsheet-sized electronic news presentation to the reader. We hypothesize that the use of "paper-quality" displays coupled with computational access to news data streams will give us a range of new possibilities in content selection, imagery, typography, and human interaction. While we are using the newspaper as a model of data access, we are more generally exploring the connectivity of user modeling, media manipulation and dynamic displays. We feel that news systems should be personalized, update continuously, provide relevant feedback and have multimedia capabilities.

1.1 Signals With a Sense of Themselves

The transformation of traditional mass media to their electronic counterparts has the potential to impart radical change upon these media. However, there is little evidence that the simple transformation of media from traditional to digital form is a critical transformation. The distinction between digital and analog forms of media is in and of itself insignificant.

In the case of a newspaper, taking the ink off of the paper and sending it as bits is attractive; it is cost effective in terms of trees and trucks. The bits can be reassembled in the receiver, and displayed as it would have originally appeared. The final display could once again be ink on paper, such as a "Fax" paper, or displayed

electronically [Gifford et al. 85]. The essence of such a system is efficiency of distribution. Little would change in the use or content of the newspaper due to digital distribution.

Of interest is the encoding of the information and how it is made available. We are interested in developing communication systems in which the digital nature of the channel impacts not only the distribution of the signal, but the content of the signal as well. Digital media affords us the opportunity, for the first time, to design a signal in which the content, and not just the packaging or encoding, is made accessible. We refer to this as a "signal with a sense of itself." [Lippman 91] Such signals are not directed at a human recipient, but rather, to a local computational agent acting on her behalf. In response to instructions from *both* the distributor of the signal and the reader, this agent operates upon the signal in manners both suggestive of traditional media and of new forms. Digital electronic media should give us the opportunity for forms of data distribution that invites user participation with the data.

To date, content sensitive signals have been applied to applications such as video compression, scene widening [McLean 91], ambient sound reconstruction [Vercoe et al. 91] and office information [Lai et al. 88]. We are interested in how such signals can be of utility in facilitating user access to large, rapidly changing data streams such as news. In our work, we concern ourselves with the manifestation of the computational access to the data.

We are using a broad range of data streams in our application development: traditional news wire-services, television news, community papers, and electronic mail. These sources vary in content from international to local to personal issues. They vary from text services, which are directly amenable to computer manipulation, to a variety of charts, maps, graphics, video and audio, which are immutable in their usual form. We enrich this data by incorporating non-lexical information, such as intonation [Hu 87], stress, pauses, type of shot, etc., as well as more traditional information, such as source, date and byline. Data manipulation takes the form of augmentation of stories by compositing pertinent text, stills and audio. In addition, we attempt to "transcode" between media types in order to account for data/display capability mismatches, personal preference and sensory impairment. The data streams are assembled at the display, at the time of display.

1.2 Paperlike Displays

Every newspaper in the world has a front page, with all that implies: content, headlines, emphasis and juxtapositions. Ironically, these features are almost universally absent from on-line access to information. Recent advances in display technology include the development of both large and flat displays: CRT technology has already achieved 300 dpi resolution; flat screen technologies are making rapid strides, mostly due to the popularity of portable displays and the perceived attractiveness of a large flat screen television. The availability of high quality local printing does not restrict electronic access to electronic displays. Such technologies enable one to rethink the way in which electronic information is accessed. One no longer has to sacrifice "front page" amenities when accessing on-line information on displays with sufficient real estate. Since both CRT and LCD technologies enable the intermingling of text with other media, such as television, radio and electronic mail, data streams no longer have to be designed with a predetermined destination. At times, large and dynamic is appropriate, while at other times, small and portable is more suitable to the user. In designing our presentation systems, our goal is to let the user, rather than the data provider, find a fitting balance between static and dynamic, large and portable displays.

1.3 Personal Computing

We postulate that news systems should provide news based upon the user's interests. They should also take into account the user's knowledge level and style preferences. This is manifested in the detail and complexity of the article chosen as well as the media types used in the presentation. Our work in personalization is based upon developing an implicit user model. This model is constructed on the premise that the user is already engaged in a rich computing environment, where her calendar, correspondence, location, and the like are made accessible to the system.

1.4 Newspace

We are developing a framework for our work in electronic publishing. The application, Newspace, described below is the current instantiation of a set of utilities we are applying to the exploration of computational access

to media. The application has three components: media manipulation, user modeling, and presentation. Communication between the components is handled by a distributed system. The basic flow of information through the system is shown in the figure.

The application described below is the current instantiation of our work in electronic publishing. The application has three components: media manipulation, user modeling, and presentation. Communication between the components is handled by a distributed system. The basic flow of information through the system is shown in the figure.

In summary, we are exploring how can we use computing and intelligent signals to provide new information and entertainment possibilities. We are exploring what opportunities and bottlenecks are imposed upon new systems of communication by computer environments which are traditionally serving a professional audience with much more directed goals and well defined boundaries regarding the scope of the data.

1.5 Paper Organization

This paper is organized as a series of discussions about the various system components. The intent of these discussions is to define the framework for our investigations. Each discussion includes a description of current implementation details. The topics of discussion are presentation, data organization, user modeling, media manipulation, and the distributed system framework.

In our discussion of Newspace, we attempt to elucidate how the UNIX environment both helped and hindered our development, how this work relates to other UNIX information retrieval systems, and to what degree the thread of our work can be considered as a general UNIX utility.

2.0 Data Sources, Data Structures

Many news sources are available as feeds to the electronic newspaper. These sources are in different media, and are transmitted in different ways, but are all converted to a common format for use by the electronic newspaper. Of critical importance is that this format allows computational entities a choice: to treat "articles" the same regardless of media, or to treat them in a media-dependent manner.

News is unlike most other data found on a workstation for two reasons. First, it is constantly changing, and by definition unpredictably—if it were deterministic, it wouldn't be news. Second, it reaches the workstation packaged in many different formats. Critical to the success of the integration of news into a workstation environment is that the workstation not be limited by either the content or the packaging of the news. Only when the temporal constraints, protocol incompatibilities, media limitations, and data structure inflexibilities are removed can an electronic newspaper be made palatable for the workstation user.

The data sources available to Newspace vary in the size of the intended audience. The scope ranges from international to local to personal; this variety of scope is most evident in the different lexical sources.

2.1 Lexical Sources

Natural language comprehension is far off. But even in the absence of explicit cues, text always allows some meaning to be extracted, unlike other media. Techniques as simple as the keyword analysis used by Newspace can provide a reasonable first approximation to sophisticated story comprehension.

The international news sources provide the highest volume of news. The Associated Press and New York Times news wires are encrypted and broadcast on a local radio station subcarrier. This signal is received by hardware which transfers the data through a serial line to an IBM XT. The XT decrypts the data and stores it using PC-NFS on a disk on our local network. The United Press International news wire is provided by ClariNet, an electronic publishing service that provides articles via NNTP over USENET. The Dow Jones News Services, which include the full Wall Street Journal and financial wires, are transmitted over a dedicated phone line using the X.25 protocol. Information is received continuously, as would be expected for data as timely as stock quotations. The transcripts of ABC, NBC, CBS, and CNN nightly news broadcasts are retrieved from Burrelle's Broadcast Database Service.

There are four regional sources available to Newspace. Three Cambridge college newspapers are available: The two MIT newspapers, The Tech and Tech Talk, are provided via UNIX filesystems accessible from the

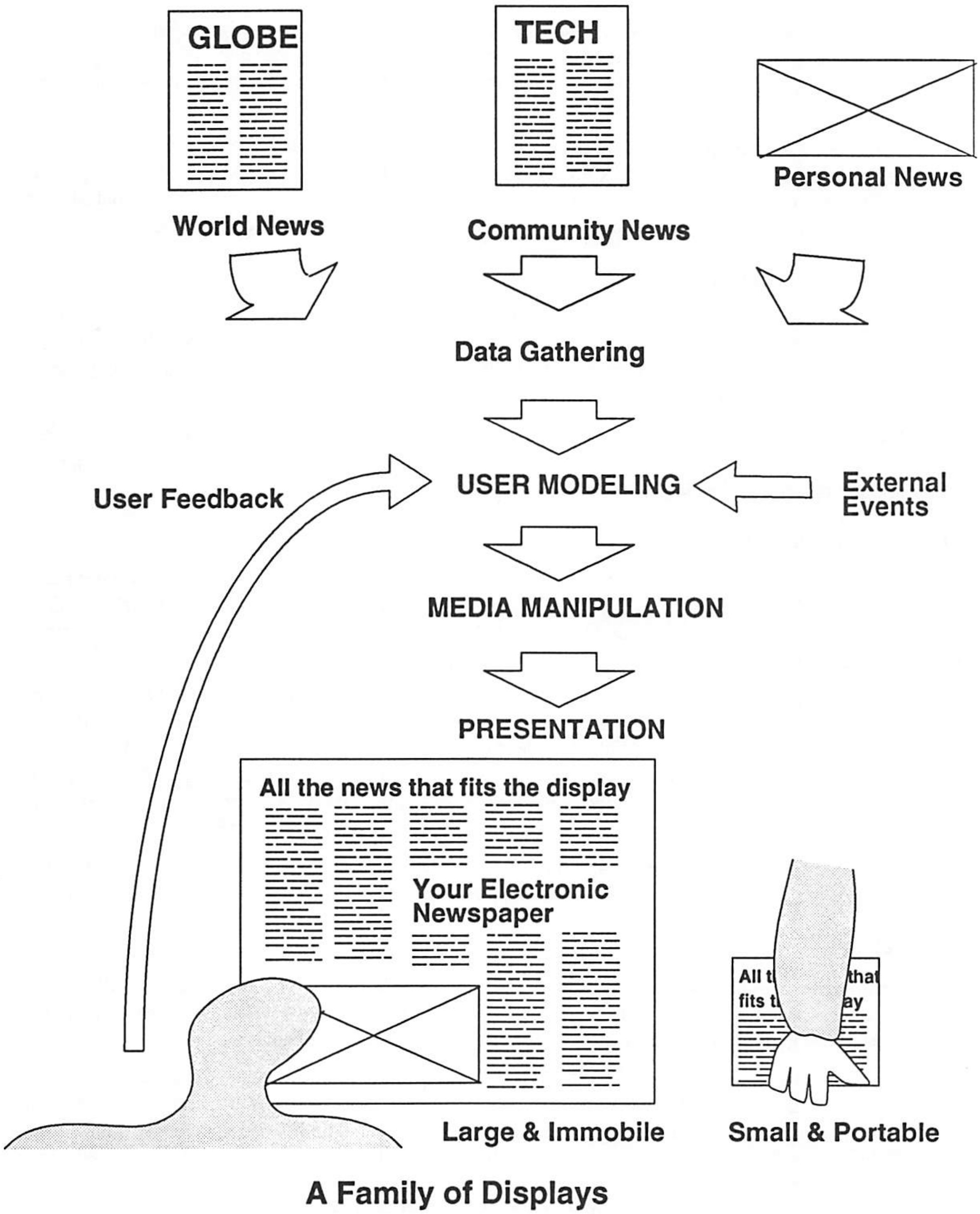


Figure 1: Once data streams are converted to a common format, rudimentary content analysis is performed, decomposing the data into objects. The data is passed to the user model, which presents a subset of the news to the media manipulation module, which in turn provides the display with a coherent, complete presentation. User interaction with the display module, as well events external to the news application provide feedback to the user model.

Media Lab. The third newspaper is the Harvard Independent, transmitted through a modem connection. Finally, local weather information is obtained by *fingering* MIT databases which supply local weather reports and forecasts, updated hourly, in a *.plan* file.

The personal source is electronic mail, which has a small but very devoted audience. A Newspace program filters a user's mail by identifying if the mail is personal, i.e. directed to the user and not to a larger group. The more intimate the mail, and the greater the importance of the sender, the more prominent the placement of the mail on the screen.

2.2 Non-lexical Sources

Currently, the only way non-lexical sources can be manipulated is to assume some lexical "slug" that describes the contents. The nature of these slugs, and what to do when they aren't present, is discussed in Chapter 4. Below is a description of the sources posing these problems for us in Newspace.

Services that provide graphics databases are becoming more common. These services maintain libraries of still pictures for use by news services. One bulletin board contains line art Associated Press graphics. The Knight-Ridder service Presslink contains several graphics libraries. These libraries utilize several different formats, all designed for personal computers, e.g. MacDraw; considerable effort is being expended trying to convert these to Postscript or other UNIX-manipulable formats.

Broadcast news is a rich data source. Video clips are digitized from television news broadcasts. The audio is kept in a separate file and synchronized at the display. We have all the elements to automate the process; we could, for example, record the seven o'clock news lead story every night. This currently is not done because of resource limitations that we hope to resolve soon.

A closed-caption decoder provides lexical cues to the content of the video. The text of the broadcast is transmitted during the vertical blanking interval; this signal is decoded and displayed on the television screen simultaneously. A decoder was modified so that the text is buffered and relayed to a standard RS-232 serial data port, giving Newspace access to the information.

Because of the overhead of non-lexical media, few regional or personal non-lexical databases exist. A face database is used to augment electronic mail. When electronic mail is sent, users can include an RFC-822 compatible keyword-token pair which is a pointer to a file containing an appropriate facial expression. The sender may choose from a variety of standard expressions: talking, thinking, smiling, or with no expression. Voice mail poses interesting challenges for inclusion into Newspace and is something we may consider in the future.

2.3 Data Types

With the exception of audio, all news is kept in a common format: the datfile. The datfile was designed to be continually changing; programmers are encouraged to build their own backward-compatible extensions. Originally designed as an image format for common storage and interchange [Musicus 88], it has been extended to handle text. Each datfile is actually a directory containing a data file and a descriptor, and optionally other datfiles. The data file contains raw image or text contents; the descriptor is a series of keyword-value pairs that describe the data. The descriptors of text datfiles include some mandatory keywords such as source, category, author, title, subject, and number of characters. Each datfile can contain links to other datfiles; for instance, a text article with an associated picture will contain the filename of the picture in the descriptor.

Each text datfile contains at least three subsidiary datfiles. These contain lists of paragraph offsets, sentence offsets, and word histograms. The first two allow flexible indexing for the presentation; the third allows a simplistic analysis of the article contents as well as a method for eliminating redundant articles. We are in the process of supporting additional subsidiary datfiles that will handle format information such as font type and size.

The subsidiary datfiles facilitate both semantic and syntactic information retrieval. For text, we can immediately retrieve any paragraph or sentence, and can perform some rudimentary content analysis from the

histogram datfile. We anticipate more sophisticated analysis techniques in the future; the flexible structure of the datfile permits such extensions.

While the datfile structure provides a conceptually easy way to manipulate articles, it can be awkward because of the number of inodes used. With paragraph, sentence, and histogram information, each article occupies twelve inodes. The thousands of new articles every day stored in this structure strains the NeXT 68040 used for storage.

Because of the diversity of formats of the news sources, each source requires its own software handler which performs two primary tasks: segmentations of the raw data from the descriptor information, and ensuring that entries to be put into the descriptor file have common keywords. As an example, one source might have an "author" keyword while another has a "by" keyword, both denoting the writer of the article. The handler converts the "by" to an "author". Each handler stores its articles in the news depot. The temporal granularity of each handler matches the frequency with which the source provides new articles. For some sources, this is more than one per minute; for others, it is once a week.

A daemon periodically checks the depot to see if new articles have arrived. It converts the articles to the datfile format and stores them in a directory that uniquely identifies the news source and the day of the month. This source based storage is inefficient. Ideally, each article should be compared to all related articles so that redundant articles may be eliminated and topically similar articles can be grouped together. This will be done in the near future and is a good example of how the digital processing of news allows manipulation based on content instead of packaging.

2.4 Document Retrieval

The storage and retrieval of large quantities of articles raises a wealth of issues in the area of document retrieval. Newspace is not intended to solve, or in most cases even address, these issues.

Newspace makes it evident that a desirable feature of information retrieval systems is relevance feedback. A story in isolation is all but useless; the user should be able to view the article in context through the presentation of related articles. Two styles of relevance are identified in Newspace. The first can be viewed as "stepping back and taking a look around"—more articles at the same level of analysis and about similar topics. The second is a more in-depth analysis of a particular story. Both of these referencing methods are supported by a master index that maintains a keyword summary of all articles stored in the database. Due to storage limitations and inefficiencies, we are limited to two days worth of news, amounting to 300 megabytes of text.

Suppose the user takes a week-long vacation. When she returns, her newspaper should display the most important news of the week. For an electronic newspaper to accommodate her, it must be able to store gigabytes of news. News agencies store immense quantities of press files and photos so that when a story breaks, they are prepared with background information. This also requires vast amounts of storage.

Issues of document retrieval at this scale are addressed by Thinking Machines' Wide Area Information Server, developed as a joint project between Apple Computer, Thinking Machines, and Dow Jones [Kahle 89]. As personal workstations grow in power, they can find, select, and present documents locally to tailor to the users interests and preferences. However, the workstation should not be expected to hold all the data needed by a workstation. Wide Area Information Servers answer queries over a network, providing either pointers to documents or the documents themselves for personal workstations or other servers.

The Wide Area Information Server provides a robust framework for the maintenance of a large central database of news. Newspace could make good use of such a server. Unfortunately, its protocol restricts it to text and so does not address the multimedia issues, but it provides a glimpse of what will happen when we start to store and access the immense quantities of information that any newspaper requires.

3.0 User Modeling in the Electronic Newspaper

An important benefit of the computational manipulation of news is that the newspaper can be different for each person. Both content and presentation are tailored to each individual. This personalization is

accomplished through user modeling—maintaining databases about individual interests, plans, beliefs, behaviors, misconceptions and schedules. The models are used primarily for article selection, but can readily facilitate any sort of customization.

The models of individuals are constructed with a user modeling system. This system, Doppelgänger [Orwant 91], collects information about a population of users, makes inferences from this information, and provides the results for clients. The electronic newspaper is the primary client.

Doppelgänger user models are dynamic, changing both as the system learns more about the user and as the user herself changes. The data it maintains include objective (sex: male) and subjective (bias: liberal) information as well as long-term (occupation: researcher) and short-term (idle time: 2 minutes) information. In addition, there is also implicit information inferred from the universe of user models: generalizations about populations of users are made and used to establish default assumptions about user models. By monitoring user actions, including but not limited to the Newspace interface, the system bolsters both precision and accuracy of the model over time.

Although the model is meant to be as comprehensive as possible, the data gathering is kept unintrusive. Other user modeling systems ignore or sidestep this problem of data acquisition by either assuming that the information is provided to them by the user, e.g. by a questionnaire, or by categorizing users into preset stereotypes. Neither alternative is acceptable for our purposes. The burden of enforced questionnaires would make the system un attractive to the user, and preprogrammed stereotypes lack the flexibility needed for adaptation to the breadth, depth, and variability of user preferences.

3.1 User Modeling in a UNIX Environment

The concept of the user model as an immense store of data about the user presupposes computational accessibility to this data. To a large extent, then, the extent of the model depends on what information UNIX can provide about users. Electronic mail, calendar, last, lastcomm, the aliases file, and finger are all conventional UNIX utilities that, when interpreted properly, provide a wealth of information about users. The Badger system, in use at the Media Lab, tracks the physical location of participants wearing badges. These badges emit infrared bit sequences that serve to identify the physical locations of their wearers.

All these sources of information about the user supplement the most important data source: the electronic newspaper itself. The system's prediction of what the user will read is compared with what the user actually reads. The user's choice of which articles to read and which to ignore provide important feedback on the success of the user model. Most of the information gathered in the user model can be applied in one way or another to affect the content and presentation of news.

The models are used to select articles on the basis of category, keywords, source, and timeliness. We would like to be able to make more subtle distinctions, but two obstacles prevent this: the lack of adequate story comprehension and the lack of content cues provided at the time of distribution.

While the primary purpose of user modeling in the electronic newspaper is choosing which articles to be presented to the user, other information available in UNIX and provided by Doppelgänger can affect the manner of presentation. For instance, if the user is idle, articles won't fade away, since the user won't have had a chance to read them.

Expressed interest in an article can be interpreted as a request for more information. Chapter 2 discussed how this information is retrieved. The user model maintains which articles have been read, and this information is used to ensure that the articles retrieved are both appropriate and fresh.

Articles are important to different people for different reasons, and the user model knows why. The factors contributing to the selection of an article are reflected in personalized headlines. The same article may be of importance to two readers for completely different reasons. Consider an article describing a blizzard in Boston. One reader's headline might be simply "Boston Blanketed by Blizzard," but to an Oakland reader, "Red Sox—A's Game Canceled."

3.2 Learning in the User Model

The varieties of user data maintained by Doppelgänger have different representations, each with different learning strategies. The most useful representation for the newspaper is the strength-confidence entry. These indicate user preferences for some attribute of a news article, such as a particular topic, category or news agency. Each entry has a strength, which indicates how important the preference is, and a confidence value, which is the Doppelgänger's estimate of the accuracy of the strength value.

Learning is made possible as a result of the feedback provided by the newspaper. The placement or exclusion of each article in the newspaper is the result of a large number of user model entries affecting the priority of the article. Some entries increase this value; others decrease it. The way each entry "voted" is remembered, and is used to punish or reward each entry after the user's response has been determined. A gradient descent algorithm is used to adjust both strength and confidence values for each entry that contributed to the decision to include or exclude the article from the user's perusal. A high confidence value prevents the strength from changing drastically. Conversely, a low confidence value allows the strength value to sway widely. For most entries tested so far, the strength has converged to a steady state within a few hundred iterations.

3.3 Different People have Different Models

An intriguing aspect of user model maintenance is the ability to view newspapers filtered by others' models: seeing the newspaper through their eyes. This is supported; a user can view the newspaper using another's model, or can interpolate between two models. Also of interest is the ability to extrapolate beyond two personalities, or even to caricature the user's own personality. Individual attributes can also be directly controlled. These are the first step to user manipulation of more complex qualities such as political bias or depth of analysis. Fabricated models can be constructed to produce extremist newspapers in which one or a few traits are held to be of maximal importance.

Because of the sensitive nature of the data included in the user model, each user is given ownership of her model. The model can be displayed as a list of object-attribute pairs which can be modified by the owner. The user can explicitly specify all the information in the user model, if she so desires, or everything can take place implicitly, with the system learning about the user through her interaction with the newspaper. The interface through which the user edits her model is itself a source of data for Doppelgänger, providing information about which attributes should be calculated differently.

3.4 What's Next for User Modeling

The improvement of user modeling for electronic newspapers depends upon advances in three areas: user data acquisition, content analysis, and the philosophy of news. The first is necessary to enrich the user model and will benefit all user modeling; the second is specific to article selection and is necessary to fully utilize the user model; the third will provide insights into the different reasons people read news. User modeling will play an important role in the development of effective interfaces as well as content personalization. Interface design will come to rely upon the system's ability to sense and respond to the user's activities in a meaningful way.

Data acquisition devices such as an eye tracker have been used in user modeling in the past and can provide valuable information for Newspace. As we become better able to sense the user's actions, we will also need new ways of interpreting the information. Doppelgänger is not yet capable of sophisticated insights into why people read news. Allen suggests a number of motivations for news reading [Allen 87]; some of these are dissimilar enough that the structure of the newspaper could be radically changed if we could gauge them. Content analysis is discussed further in Chapter 4.

When the full benefits of user modeling are realized, the electronic newspaper will become perfectly adaptable, providing each user with a newspaper ideally suited to her tastes. The user will be provided with all the articles he wants to see and none that he doesn't. Articles of interest to the user will be selected even if the user was not aware a priori of the articles or the reasoning behind their selection. Users will not be deprived of the serendipity that allows them to stumble upon an interesting articles. Most importantly, the modeling will gradually slip beneath the user's awareness as the user grows to trust computer control over both the content and the presentation of her newspaper.

4.0 Multiple Media Manipulation

Our news system gathers information from other media than the traditional text wire services. In this way, we can capture as many diverse sources as possible. A story of interest may only have been aired on a local public affairs television show but not in the local print. But a more compelling reason for using multiple modalities is that each form of media is inherently powerful in expressing different types of information. Quantitative information, such as the Dow Jones Average Trends, is best described graphically. The conceptual reasons behind a recent market crash may be better expounded by text. News of a speech or debate would be best presented by an audio clip, particularly if moments of tension or elation were realized verbally. Pictures and video engender a certain reality about the existence of an event or a person. They allow us travel to the scene of the famine or see the emotional state of our most recent email sender.

The digital domain provides us the ability to efficiently parse, search, compare and retrieve news information. We can then combine and transform stories in ways the editors never imagined. Linking is possible and so is the creation of composite forms. Graphics can be combined with the text or audio from a different news source. It is also possible to "transcode" information from one media source to another while retaining saliency.

4.1 Display/Personal Limitations and Transcoding

Our current display runs on a large screen platform with audio and video capabilities. While we are experimenting in using this environment to its full potential we are also considering how the system should behave in more limiting instances. For example, the laptop domain affords portability at the expense of bandwidth. This loss is manifested in a lack of audio and graphics capabilities, or in the case of some platforms, a reduced video frame rate. Yet we maintain that the user should not be denied information of interest simply because of computational limitations. Nor should the user be denied access because of personal limitations such as being visually, aurally, or lexically impaired.

We attempt to transcode information from one media type to other more displayable and understandable forms. This will be a lossy transformation in almost all cases. Ideally, however, we would like to ensure that significant information is preserved. One example of this transcoding process is the transcription of audio. Unlike the typical captioning used with television, we are using techniques to convey more of the "content" of the signal. The layout of the text can convey points of emphasis, volume, speaker changes, and perhaps even emotional speaking state [Bender et al. 88, Hu 87].

Video could be represented by a salient still frame or more succinctly by a single object or objects in the represented frame. Scene composition is feasible; a scene could be created with three objects from contextually related but distinct frames.

4.2 Representations

Computer representation of objects or concepts necessarily depends on the goal of system. Our current aim is to be successful at augmentation and transformation of news information. Ideally, the future computer will be able to synthesize multimedia news presentations for the user. Before this is possible, much more work needs to be done in the codification of narrative and rhetorical predicates and also in the description of multimedia primitives. Encouraging research in these areas is being performed in the Media Lab Music and Cognition Group and the Interactive Cinema Group respectively [Davenport 91].

4.3 Object Oriented Approach

In order to accomplish our goals, we have adopted an object oriented approach to multimedia. It is an economical and powerful representation and the objects can respond to a simple set of heuristics. Our objects are not limited by the traditional boundaries of the story. Paragraphs can be highlighted as an object and treated as such. Media can be decoupled: the audio portion and picture portion of a video segment can be deemed as separate objects and searched independently. This is ideally the representation needed to describe video with voice-over narration or music overlay.

The advantage of this method is that these objects can have some of knowledge about their contents and contexts embedded in their representation. Links between objects are possible. Some may be explicit such as "this audio object belongs to that video object." Others may be created dynamically by the system as it receives new information in its incoming news data stream— "this graphic in our database will best augment this new text story".

In addition, more textual descriptive information concerning the media signals must be stored. This granularity of the description may be at the story level: intended audience, political bias, difficulty level, related articles, associated graphics. A rudimentary implementation of this type of description is currently being broadcast in text transmissions of the AP and UPI news wires. We readily use the 'urgency' and 'associated graphic' information supplied by these sources. We need a finer level of detail to enable intelligent compositing of media. The information could be additional editorial markers such as "from byte x to byte y is an emotionally charged moment in the audio track of the debate" as well as more factual information such as audio intonation patterns and camera attributes.

4.4 Video Objects

The representation of video material presents particular difficulty since there are audio tracks associated with the video, and moving objects and moving cameras in addition to the loss of information from 3D to 2D. A manageable way to deal with this information is to describe the objects individually within the frame rather than assign descriptive attributes to a series of frames. This concept has been advocated in many fields from image compression and computer graphics data bases to archival video storage representations.

Traditionally, moving image representations for digital compression have been accomplished by differing means of encoding the entire frame as a complete unit. Hybrid video coders, subband and pyramid decomposition coders, and region coders are all examples. Conversely, model-based coding allows coding of separate objects within the scene. The transmitter and the decoder agree on the basic model of an image and the transmitter sends parameters to manipulate this model. [Watlington 87, McLean 91]. Work in this area is the inspiration and foundation behind our video database.

Media objects become their own entities but are also part of larger classifications such as 'background' or a group. Objects belong to other objects and hierarchically inherit their characteristics such as the lighting for a particular scene. 'Content' can be added to the objects—such highlighting a particularly good reaction shot or associating links. The frame can still be an object in our system but its role as the smallest video building block has been displaced. Now its prime purpose is as a spatial marker to the location of the object within a large digital storage medium.

This object based description allows us to save bandwidth in a content dependent way. Perhaps the background of the White House will be stored locally and only the newscaster and interviewee will be transmitted to the computer. These transmitted objects will have associated camera angles, gaze vectors, and lighting descriptions so that we will know how to 'doctor' the received objects at the other end to appear as a complete original scene.

In creating a larger salient picture, we must follow composition rules that do not unsettle the viewer. For example, we may want to abstract out three objects from a series of contiguous shots and make a "salient" still. During a recent national news broadcast, two individuals were discussing Saddam Hussein's desire to emulate. Each one was shown alone in the frame. They were intercut with a picture of a photograph showing Hussein and someone else. The photo was scanned from the left side to the right. In creating the composited still, we recreated the original photograph by fitting together the frame showing the left half of the picture and the frame showing the right half. The two halves were melded together using a laplacian decomposition and filtering. The two speakers when cropped from their original frames were not of proportional size. One was filtered accordingly. This resulted in the text caption of the photo being incorrectly sized. The text was then treated as an object and subsampled and filtered accordingly. Ideally, if the objects are speaking characters they should be facing each other or the object being discussed. When abstracted out of the original frame, gaze vectors will tell us if they are oriented correctly. If they are not, we can reorient the image.

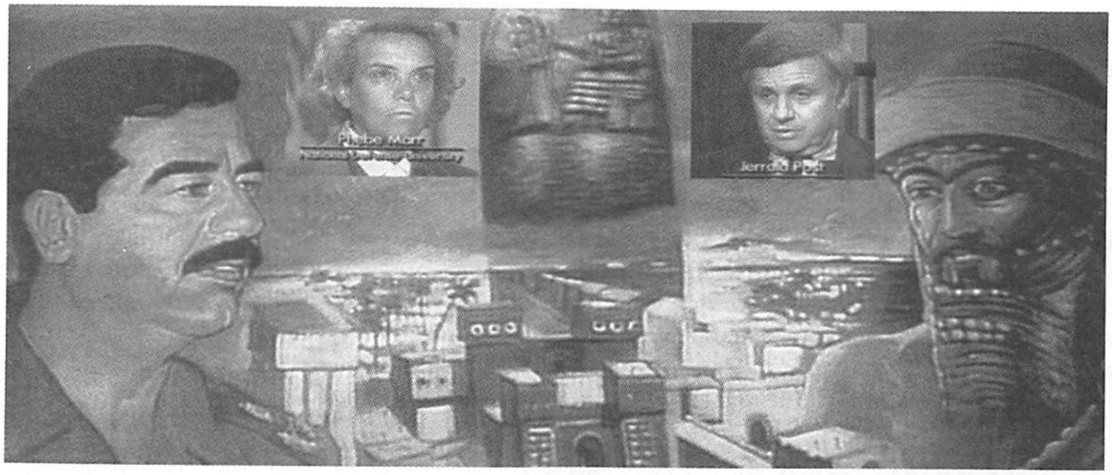


Figure 2: A salient still has been generated by compositing a sequence of frames from three television news scenes. The background image was composed by fitting together frames from a pan over a wide aspect ratio original image. The two inserts were cropped from their original frames and resized when fitted into the resultant still image.

Information needed to accomplish these manipulation include camera characteristics such as focal length; type of shot—wide, medium, close-up; movement—crane, dolly, tilt, pan, zoom. Also needed is some form of depth information such as distance from lens.

4.5 Parsing vs. Editorialized Information

Filling information into our data base can be accomplished either by having the computer parse out the information or by entering the information in manually. Until we have story understanding systems and computer vision systems, most of this work will have to be done by hand. Ideally, much of the burden of this coding could be done at the publishing source. Editors could highlight significant moments in audio, and camera information could be saved as the image is being recorded. [Davenport 91]

Our approach at this moment is a combination of the two. Keywords are extracted from the lexical content of a news story and matching is done between articles. Simple metrics are applied to determine the "difficulty" of an article: sentence length, word frequency in the english language. Topically intense areas of a passage are found by passing a window over the text and determining portions of interest to a user. Video information is encoded by hand with the exception of broadcast transcriptions.

4.6 User Models and Multimedia

Our news system is not driven by any particular media source. All are searched equally earnestly. When given a choice between differing media coverage of the same news, many factors can dictate the final choice. Information from the user model is one factor. Past interaction may show that a user prefers to read economic news but always reads sports news when given the opportunity. A user's knowledge level of a topic will also help to determine the source and media type. Studies indicate that people learn more rapidly from television news than from purely textual sources [Just 89, Neuman 88]. The user might want their news broadcast to have a particular "feel" to it: the Wall Street Journal versus Entertainment Tonight. This could change depending on the time of day or current location or activities of the user.

4.7 Virtual News—The Future

Perhaps in the near future, a user could truly personalize her news by manipulating such characteristics as who broadcasts it and where it is from. The user could always have a woman's voice broadcast the news [Hawley 91] or always have Peter Jennings as the talking head. A user could watch the news from Mr. Rogers'

point of view and perhaps even have him broadcast it from his home.

5.0 Presentation: All the News that Fits the Display

Although the Newspace architecture supports a variety of displays, the discussion below is confined to a broadsheet presentation. "Paper quality" displays give us the possibility to explore the "newspaper metaphor" within the context of a computer screen. We try to keep the best of the newspaper metaphor while taking advantage of the dynamic nature of the display and the processing capabilities of the computer.

5.1 The Newspaper Metaphor

We start by analyzing some elements in what we label the newspaper metaphor. After centuries of evolution, the newspaper format has reached a degree of stability and standardization. Despite sociological differences, readers from different parts of the world can pick up a local paper and immediately know how to read it if the written language is known. The front page is the most distinct and universal feature of the newspaper format. It was invented 300 years ago [Gürtler 84], and has changed little since then. Without exception, industry predictions of the future of the newspaper include a front page [DESIGN 88].

The front page is dominated by the headlines of the most important stories. The size of the headline font indicates the relative importance. By glancing at the front page, the reader can get an overview of the most important issues in a matter of seconds. The front page is not bereft of content. Below the headline is printed the body text of the article. Simply by changing from scanning to reading, the user is able to change modalities from overview to detail. While the front page contains the lead stories from multiple categories, inside pages are more specialized. Main categories have their own section, such as "sports" and "business." Within these sections are found additional detail, related topics and juxtapositions.

5.2 The Newspaper Interface vs. Computer Interfaces

There are fundamental differences between the newspaper interface and that of the usual electronic information retrieval system (IR). Newspapers use layout, editions, and culling by "experts" to provide a facile and forgiving interface to a database that is of only selective interest to the reader. Information retrieval is by and large sequential access wholly under direct control of the user. No assumptions are made by such systems as to the user's intent, hence the presentation of the data is not tightly coupled to its retrieval.

Newspapers update themselves without waiting for requests. In creating the front page, the newspaper provides either periodic, as in the case of editions, or continuous information to the reader. Even if a subscriber ignores the paper for a week, new editions are produced. IR is event-driven, i.e., nothing will happen unless the user issues commands, e.g., a mouse click or button press. The interface keeps the user active and in control.

The size of a broadsheet front page makes it possible to display huge amounts of information, which makes that information directly accessible. IR requires the user to initiate a search and to issue several commands, e.g. pan and scroll, to access the same amount of data.

While IR can claim interactivity, the newspaper is supposedly a medium for one way communication only. Still, by turning to a section page a reader can drastically change the content of the page being looked at. The stories on the page have become more specialized, and in one sense the reader has interacted with the editors of the newspaper. By turning to the section page, the reader requests more information, and the editor "recognizes" the request by providing more coverage.

When preparing the newspaper, and especially the front page, editors process the information to accommodate all readers. A reader that only spends a few minutes reading the news will easily pick out the main stories by scanning the headlines, which summarizes the content. Journalists write most stories in the so-called "inverted-pyramid style"—the main points of the story are described in the first paragraphs, while the last ones are written with the editor's scissors in mind.[1] Therefore, by reading a paragraph or two, the reader will comprehend a disproportionately large part of the news. Minimum effort is required to digest 10% of the information, and there is little penalty in skipping the remaining 90%.

---

1. Except narrative style articles.

5.3 The Broadsheet Presentation

The goal of the presentation subsystem is to enable the user to rapidly search large quantities of news with little or no predetermined focus. The process of browsing should help the user uncover and read articles of interest, without incurring significant overhead on the part of the user either in terms of time or effort. While the user modeling subsystem obviates the requirement that the user make explicit specification of interests, the penalty for displaying an "uninteresting" article should be minimal.

To address this goal, we organized the presentation as a two dimensional virtual plane, a "Newspace." Within this space, stories appear in topical clusters. The front page is a cluster of the "most important" stories. A map of the Newspace is provided to facilitate navigation. The remainder of this section is a discussion of an implementation of Newspace. This includes a discussion of article formatting, page (or cluster) layout, the map, and the user interface.

5.4 Presentation System Configuration

The hardware configuration consists of a Sun Sparcstation 370 that hosts a VME frame buffer from Univision, model UDC-4012. The frame buffer drives a Sony DDM-2801C monitor, which features 2000 line resolution and a 60 Hz progressively scanned refresh. The viewing area of the monitor is 20" x 20"; almost as high as a broadsheet newspaper and a little wider, but not as portable. At least three people are required to move this huge piece of glass. In addition to the primary monitor, the system also includes a second color monitor with a more conventional 1k resolution.

The presentation software has been developed within the X Window System (X11) running on the UNIX operating system [Scheifler 87]. To run X11 applications one needs a server application that stands between the client applications and the hardware. The server handles the keyboard, screen, mouse and one or more screens, while providing client applications with a standardized interface. MIT's sample server was ported to the Univision Frame Buffer [Lie 91]. This allows us to use the previously described hardware configuration as an X11 workstation and makes large amounts of existing software available.

5.5 Article Formatting

The purpose of the formatting process is to take ASCII-based text, with optional illustrations, and render it into a pixmap that can directly be displayed on a bit-mapped computer screen. The formatted "image" contains elements of the newspaper metaphor, e.g., headline, byline, columns.

A problem when formatting news articles is that not all information is available when needed. For example, when selecting which headline font to use it is, among other things, important to know how wide the article will be. To know the width of an article the formatter must know the number of columns. Before it selects the number of columns, the formatter should know what headline font is to be used so it can minimize white space areas!

Another dilemma one faces when formatting text is legibility vs. word density. Newspapers often have a high word density at the expense of legibility. A good example is the front page of the New York Times. There is a minimum of white space, and headline fonts are often condensed. Margins are minimal, and the overall impression is "dark."

All text rendered by the formatter uses antialiased fonts. [Negroponte 80, Schmandt 80, Warnock 80]. The use of antialiased fonts allow the formatter to consider the display device as continuous, rather than as a discrete matrix of pixels. While this doesn't improve resolution, it does enhance the addressability of the existing resolution. This is helpful for properly rendering letterforms, as well as positioning the letterforms on the display. Ergonomic studies show that antialiased fonts are easier to read [Bender et al. 87]. Without their use it would be much harder to claim competitiveness with paper.[1]

---

1. You don't need to be a sophisticated user to be sensitive to letter spacing miscues. The 7 yr. old daughter of one of the authors complained about the letter spacing on a high end display Postscript(r) display.

5.6 Layout

The layout of a newspaper is designed to attract readers and to optimize the newspaper's effectiveness in presenting information. Rules and conventions have evolved over the years and most newspapers share well established layout principles.

Newspaper layout was one of the first newspaper tasks for which computerized automation was attempted. The problem is reasonably constrained; the program is given a set of news articles and advertisements. The ads are placed according to one set of rules, while news articles are placed in the remaining space (the "newshole") according to another set of rules. Since Newspace does not yet contain advertisements we have left out most of the discussion concerning them.[1] Without the ads the problem is similar to the computer game Tetris; blocks are placed to minimize white space.

The electronic newspaper borrows many elements from the newspaper metaphor, including the tiled layout, but page layout is quite different. While paper-based newspapers are issued in discrete editions, the electronic newspaper is continuously receiving articles to be included on the page. Accordingly, old or unimportant articles have to be removed and this complicates the shape of the newshole. This is a dynamic page and not your average Tetris game!

The broadsheet has one front page and several section pages. The sections follow traditional layout rules and tile the articles. Each page is laid out in a 5x8 grid. Space in the grid is allocated as stories come in.

The formatter produces rectangular articles with text displayed in full. This avoids jumps from the front page to an inside page, which is shown to lose one out of five readers [Bush 68]. However, real estate on the front page is limited. There is not room to show many articles in full. There are many approaches to the front page real estate problem. One solution would be to format two versions of the article; one intended for the front page and one for the section page. This approach is currently beyond the capabilities of our article formatter. Another approach is to format the whole article, but only display parts of it at any one time. Both of the above solutions would hide parts of the article from the user while reading. As long as it is possible to fit the article on a page, we believe it should be shown in full.

We have implemented several alternative approaches to the front page. One is to stack articles using overlapping windows. Important stories float to the top of the stack, less important articles may be partly visible, and the articles with the least priority are completely covered. By selecting an article, the user can cause it to float to the top of the stack, where it is shown in full. This front page contains much information and looks chaotic at times. See figure 3.

Another approach is to make all stories appear on the front page initially and have them migrate into clusters in the section pages. The migration is based upon both the article content and the goals established by the user's interests. The front page is the point of dispatch. A problem with this method is that articles are on the front page only for a short time before they are pushed into the section pages.

5.7 Maps

The broadsheet is large enough to require some navigational aids. We have chosen to exploit the potential of the map to serve as an aid. There is much precedent in this choice, including the Spatial Data Management System at MIT [Bolt 84] and the Rooms Project at Xerox PARC [Henderson 1986]. More recently, maps have become a popular window system adjunct.

Several different map schemes has been implemented for Newspace. The hardest decision we faced in designing the map was how to make both a tractable and useful abstraction of the space. Since the space is occupied principally by articles, the problem can be restated as how can we automate the creation of iconified versions of the articles. The headline of an article is a useful and readily available abstraction of an article. But simply rendering the headline gives no indication of the structure of the article, e.g. where the figures are, and how many paragraphs there are. A decimated version of the article pixmap is indicative of structure, but at a scale of at least 7:1, the decimated headline is usually illegible. Our solution is a hybrid of both alternatives. It uses a decimated "photo" of the original article upon which it renders the headline. To improve

1. Advertising is essential, both to make it economically feasible, and to enrich the content.

Figure 3: One way of organizing an electronic front page—articles are stacked using overlapping windows. Important stories float to the top of the stack, less important articles may be partly visible, and the articles with the least priority are completely covered

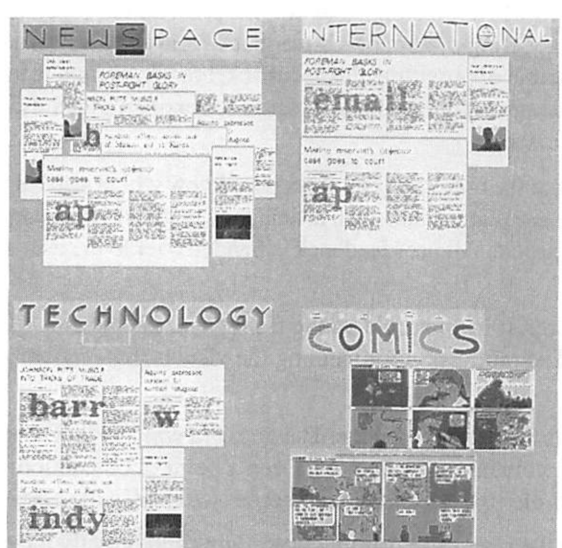


Figure 4: The second screen contains a map of the Newspace.

legibility, proportionally larger font sizes are used to render the headlines. This solution is reminiscent of the satellite photos used in weather reports where state lines are rendered on top of a photomap. See figure 4.

During the last year, the X11 community has seen the introduction of several so-called "virtual window managers." The window managers allow X11 displays to use a virtual plane larger than the physical screen size. The user can pan the real screen over the virtual screen to view a different part of the plane. The user interface for all the virtual window managers is implemented through a map. The user can see the outlines of all top level windows in a special window that is an isomorphic representation of the windows on the virtual plane. The implication of a virtual window manager to the users is that they have more room in which to place windows.

The Newspace Virtual Tab Window Manager (NVTWM) was developed to manage windows in the newspace. It is based upon VTWM [Edmondson 90]. The current release of VTWM creates a map of the virtual plane according to the user's specifications of scale and position. If VTWM manages more than one screen, each screen has its own map. Since it was feasible for the project to use a second screen for the map, VTWM had to be modified to allow a map from one screen to be displayed on another.

VTWM's representation of windows in the map consists of an isomorph rectangular area (technically an X11 window), optionally equipped with a name label. The user can specify background color and a label font. While this might be sufficient information in a programming environment where windows seldom are deleted or created, it is not sufficient in a news environment where new news comes in every minute. One way of increasing the information content in the window representations is to change the background pixmap. X11 already provides the functionality for clients to indicate icon pixmaps. While the use of icons is one way to manage a limited screen area, the virtual desktop model replaces traditional icons for most users. Using the icon pixmap functionality to set the background of the window representation was a natural modification to VTWM.

5.8 User Interaction and Participation

The bulk of this section is dedicated to the presentation of information; user input is of secondary concern. There are several reasons for this. While most software today is event-driven, the electronic broadsheet is a continuous process that can run without any user involvement. The newspaper will update itself, just as a traditional newspaper is delivered to subscribers whether they read yesterday's edition or not.

The presentation system is very much dependent upon user modeling to make sure the presentation is properly focused. The model is consulted when selecting what news to present. However, to adapt to a user's changing interests and habits, the user modeling system needs feedback from the user's electronic broadsheet actions, necessitating input devices capable of providing this feedback. Ideally, system feedback should be transparent to the user. In the current implementation, we have to settle for explicit methods; the current configuration includes a mouse as pointer device.

To trace the changing interests of the user, the system needs to know which articles the reader finds interesting. Users are encouraged to move the pointer into each article they read, and they can indicate special interest in an article by "clicking" in the article. All user feedback is handed over to the user modeling module to update the personalized filters.

If the user indicates special interest in an article on the front page, the system will pan the view into the corresponding section page. The motivation for panning the view is that readers will probably be interested in reading related stories.

All section pages contain a "door" to the front page. The door is a window that when clicked will pan the view back to the front page. The placement of the door has been a point of contention; some users favor a fixed position on all pages; others prefer the door along the border with, or in the corner closest to, the front page.

A reader can always override the automatic panning procedures and directly move to any page in the paper. The map accepts mouse clicks and will pan the view to the corresponding page.. To click in the map, the user has to move the cursor over the edge of one display to enter the other one.

6.0 A Distributed Communications System

News presentation is an inherently dynamic and distributed process. Multiple processes are required to run in parallel in order to maintain a clean division of responsibility between different modules in the system. Clearly, the news system requires a good communication system in order to dynamically transfer information between separate modules. To facilitate this communication, a package for handling communications in a distributed system was written.

The design of Newspace drove the design of the distributed system. Newspace was designed as a set of modules which had to talk to each other. Lines of communication were described between the modules. All this was done without any assumptions about the implementation of the communication network. Thus, the concepts of servers, clients, and other IPC concepts were not specifically included in the initial design. The belief was that the communication scheme should fall out intuitively from the design of the system, and the communication system should be tailored to the specific needs of the system.

What actually happened was a compromise. Inspired by the initial sketches of the system design, a generalized distributed system package, called Dsys, was created. This package kept the intuitive feel of establishing arbitrary connections between different processes, but also required that the design of Newspace be revised to make more rigorous specification of its communication needs. The concepts of server and client were retained, but took on watered down meanings. Traditional views of server/client models imply a large gap between servers and clients, in terms of function, computational power, resources, etc. While Dsys does not preclude these assumptions, they are deliberately avoided in the design of Dsys. Instead, the only difference between server and client is that the server waits for a connection, and the client initiates a connection. Thus, the Newspace designers only had to adjust their communication design to decide when their modules would take on the roles of server, client, or both.

Simplicity was critical to the design of Dsys. Dsys was designed to be a package which would allow the transformation of a system from blocks and arrows on paper to actual code in as little time as possible. As a result, Dsys provides a minimal interface to the programmer, while retaining a firm base of functionality suitable to many applications. Most details are hidden from the programmer, so one need not face a plethora of options and features when attempting to use a particular function. The "only one way to do it" approach resulted in a very short startup time for incorporating communications into the existing news system. It also eliminated most of the confusion that programmers have when trying to decide how their processes will communicate; with all the details hidden, the programmers only had to concentrate on when and what kind of data they would send back and forth.

6.1 Design of the Distributed System Package

Dsys is built on top of the TCP/IP package [Leffler 86] which means that it inherits most of the problems and features of that protocol. TCP/IP is built for establishing reliable, two-way links between processes. It is not suitable for one-way broadcasts, nor is it necessarily a good choice for developing time-critical applications which push the limits of the network bandwidth, such as a digital video server. Dsys also has these properties. Dsys is well suited to developing distributed systems where modules can communicate in private two-way conversations, and where microsecond timing is not critical. If facets of the application do not fit this model, it is best to use some other method than Dsys to handle these particular functions. This is done in the Newspace system. For most of its general communications, such as ASCII text transfers, event notifications, and simple synchronization, Dsys is suitable. However, certain news presentations require video to be transferred over the network in real time, often synchronized with audio, and Dsys does not work well for this. Instead, separate movie and audio servers are used which rely on their own protocols to achieve the required video rates and synchronization. Even in these cases, the movie and audio servers use Dsys to know when and to whom they are to serve their movies.

Dsys is based on objects called "stations". These objects are the agents which initiate and receive connections, and communicate with each other along these connections. There are two kinds of stations: server stations and client stations. A client station is a simple connection mechanism which can only initiate a connection to a remote server station. Conversely, a server station waits for connections to be initiated by remote client

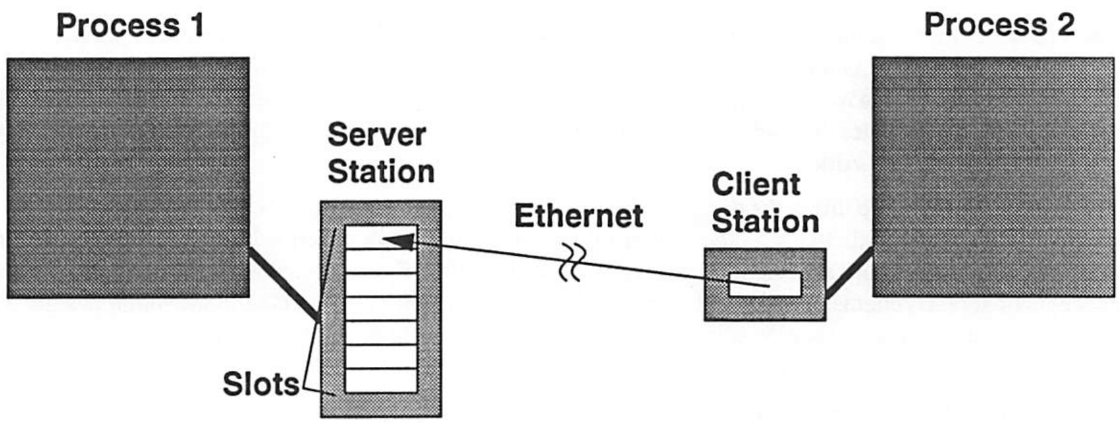


Figure 5: Client station connecting to a server station

stations. Once a connection is established, it is symmetric, i.e., clients and servers send and receive data in the same way.

Each station contains "slots" which are the connection points of the station. The station can maintain one connection for each slot. Client stations only have one slot; they exist only to connect to a single server, and are destroyed when that connection is broken. Server stations, however, can have many slots, allowing multiple client stations to connect to a single server station. Slots in the server station are dynamically allocated and freed as client stations connect and disconnect to the server station.

By concentrating all communications functions into client and server station objects, Dsys avoids the strict assignment of processes into roles as clients and servers. Instead, a single process can maintain a mixture of client and server stations, which provides the flexibility for which Dsys was designed. In the design stage of a distributed system, designers may not be thinking in terms of servers and clients when assigning the roles of the modules and communication lines. Therefore, the final design will probably contain modules that can not be categorized strictly as servers or clients, but may be playing the roles of both.

7.0 Conclusion

Mass media and personal computing are on a collision course. When the Newspaper Industry does become the Electronic Publishing Industry, there is an opportunity for it to refashion itself. The intersection of mass media, personal computing and modern communications systems portents changes to much more than the mechanisms of news distribution. When this distribution is directed at the personal computer, the ultimate interpretation of the information by the reader can be enhanced by localized and personalized filtering and formatting. The computational intermediary, acting in concert and on behalf of the "reader" makes possible new publishing styles and forms. Mass Media no longer need be monolithic, impersonal, synchronous, colloquial or prepackaged. Rather, it can be redefined to be distributed, responsive to personal needs and interests, timely, international and presented in a dynamic form.

8.0 Acknowledgments

The work was supported in part by IBM, Inc. The authors would like to thank Pascal Chesnais and his gang of UROP students for endless help and advice on most all aspects of this project. We would also like to thank Patrick McLean for his help in creating the "salient" still.

9.0 References

Allen, Robert B.: *Selecting Articles from an Electronic Newspaper: Some Limitations of User Models*, Bellcore Computer and Information Science No. 1, 1987.

Bender, Walter; Chesnais, Pascal: *Network Plus*, Paper presented at SPSE Electronic Imaging Devices and Systems Symposium, Los Angeles, January 1988.

Bender, Walter; Crespo, Ruth A.; Kennedy, Peter J.; Oakley, Richard: *CRT Typeface Design and Evaluation,* Human Factors, 1987.

Bolt, R.: *The Human Interface,* Van Nostrand Reinhold, 1984.

Bush, Chlinton R.: The research results were reported by Chlinton R. Bush in *News Research for better Newspapers* Volume 3 1968, ANPAF. Research performed by Carl J Nelson Research, Inc.

Davenport, Glorianna;Pincever,Natalio;Smith, Thomas A. : *Cinematic Primitives for Multimedia: Toward a more profound intersection of cinematic knowledge and computer science representation*, 1991 IEEE Special Issue on multimedia. to be published.

DESIGN: Newspaper Design—2000 and Beyond, American Press Institute, 1988.

Gifford, David K.; Lucassen, John M.; Berlin, Stephen T.: *The Application of Digital Broadcast Communication to Large Scale Information Systems, IEEE Journal on Selected Areas in Comminications,* VOL. SAC-3, No. 3, May 1985.

Gürtler, André: *The history of newspaper design*, Swiss Typographic Journal, 1984.

Hawley, Michael: *The "transvoicetite" and general source-oriented sound transforms*, available through the MIT Media Lab - Music and Cognition Group, 1991.

Henderson Jr., D. A.; Card, S. K.: *Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface*, ACM Transactions on Graphics, Vol. 5, No. 3, July 1986.

Hu, A.: *Automatic Emphasis Detection in Fluent Speech with Transcription,* SB Thesis, MIT 1987.

Just, Marion R.; Neuman, W. Russell; Crigler, Ann N.: *Who Learns What From The News: Attentive Publics Versus a Cognitive Elite.* Annual Meeting of the American Political Science Association, 1989.

Kahle, Brewster: *Wide Area Information Server Concepts*, Thinking Machines Corporation, 1989.

Lai, Kum-Yew, Malone, T. W., Yu, Keh-Chaing:Object Lens: *A Spreadsheet for Cooperative Work,* ACM Transactions on Office Information Systems, Vol. 6, No. 4, 1988.

Leffler, Samuel J. et al..: *An Advanced 4.3(BSD) Interprocess Communication Tutorial. Unix Programmer's Manual Supplementary Documents1*, 1982.

Lie, Håkon W.: *The Electronic Broadsheet—all the news that fits the display,* SM Thesis, MIT Media Laboratory 1991.

Lippman, Andrew: Lecture Notes, MIT ILP Symposium, 1991.

McLean, Patrick: *Structured Video Coding*, SM Thesis, MIT Media Laboratory, 1991.

Musicus, Bruce R.; Dove, Webster; Hofmann, Bill; Isnardi, Michael; Wang, John: *Dat manual pages*, Massachusetts Institute of Technology CIPG and DSPG groups. 1986.

Negroponte , Nicholas: *Soft Fonts*, Proceedings Society for Information Display, 1980.

Neuman, W. Russell: *Knowledge, Opinion and the News: The Calculus of Political Learning*, Annual Meeting of the American Political Science Association, 1988.

Orwant, Jonathan L: *Doppelgänger: A User Modeling System*, SB Thesis, MIT Department of Electrical Engineering and Computer Science, 1991.

Scheifler, R. W.; Gettys, J.: *The X Window System*, ACM Transactions on Graphics, Vol. 5, No. 2, 1987.

Schmandt, Chrisopher: *Soft Typography,* Information Processing 1980, Proceedings of IFIPS.

Vercoe, Barry; Gardner, Bill.: *Mutilchannel Feedback Calculations in Artificial Acoustic Ambience Systems.*, Submitted to the ICMC, October 1991.

Watlington, John: *Synthetic Movies*, SM Thesis, MIT Media Laboratory, 1987.

Warnock, J. E.: *The display of characters using gray level sample arrays*, ACM Computer Graphics Vol. 14, No. 3, 1980.

Jonathan L. Orwant will be a Research Assistant for the MIT Media Lab Electronic Publishing Group starting September 1991. His research focus is user modeling. Additional interests include electronic newspapers, image processing, and color research. He is a member of the IEEE, ACM, and the League for Programming Freedom.

Jon Orwant is interested in making computer privacy violations REALLY frightening.

Laura Teodosio is a Research Assistant in the MIT Media Laboratory Electronic Publishing Group. Laura received a Bachelor of Arts from Yale University in Engineering Sciences and is interested in intelligent multimedia, user interfaces, electronic mass media, and interactive narrative. Other interests include video representation, and politics and the electronic newspaper.

Nathan Abramson is a Research Assistant at the MIT Media Laboratory, working for Walter Bender in the Electronic Publishing Group. Nathan received his Bachelor of Science degree in Computer Science from M.I.T. in 1990 and is currently working for his Master's degree in Media Arts and Sciences. Nathan's research interests include designing and implementing interactive multimedia applications built from distributed digital video and audio network servers. Nathan's current work is focused on building these digital video and audio servers.

Nathan is the leader of the most fearsome rock and roll band to mix atonal modulation with serialist progressions. His friends call him "Tater", and greatly admire the bloodstains on his guitar.

Hakon Lie is a graduate student and Research Assistant in the Electronic Publishing Group, MIT Media Laboratory. Hakon received undergraduate degrees in computer science from Oestfold Regional College, Halden, Norway and West Georgia College. His current research interests include electronic information distribution and presentation, distributed graphics environments and virtual reality. He is a member of the League for Programming Freedom.

Hakon [pronouced hawk-ohn] enjoys magnetic fields and open screen space.

Walter Bender is Director of the Electronic Publishing Group at MIT's Media Laboratory. Mr. Bender received his undergraduate degree in Visual and Environmental Studies from Harvard University in 1977 and his Master of Science from MIT in 1981. He has been hanging out at MIT ever since.

Walter is captain of the Shadow Masks softball team, perennial bottom of the Bucket League at MIT.

The MIT Media Laboratory

Glorianna Davenport
MIT Media Lab

A sound and light tour of the facility at the 6-year mark: the people, research, and progress which mark this year's vision of the future.

A short description of today's group, faculty, computer power and other factors contrasts with the historical perspective gained from anecdotes of earlier days.

We look at several projects: The Kinematic Roach developed in the animation group, Computer Graphics, recent sound analysis for machine accompaniment of live performers, Elastic Boston, a multimedia map hosting a multimedia database, and Grinning Evil Death, a computer graphics piece for which several new tools were developed. These illustrate the interactivity of groups in the lab and how knowledge form one domain informs another.

Integrating Real-Time Video with Sun Workstations

Jennifer Overholt
Dave Berry
Sun Microsystems

This multimedia presentation illustrates how video is being integrated with Sun workstations. We'll explore what real-time video means and how people are using it today in combination with other media—audio, text, and graphics. We'll highlight the hardware and software enabling technologies used to create these applications.

The presentation will also cover networked video, using Sun's VideoPix as an example of what is available today. We'll describe the video server and some experimental results in using it in a networked environment. We'll also discuss the benefits of compression and some of the emerging compression standards.

Design considerations for JPEG Video and Synchronized Audio in a Unix Workstation Environment

Bernard I. Szabo, Gregory K. Wallace
Digital Equipment Corporation
szabo@gauss.enet.dec.com, wallace@gauss.enet.dec.com

Abstract

This article describes design considerations for use of the draft ISO JPEG standard as a digital video compression technique, to record and play back motion images with synchronized compressed audio in a Unix-based workstation. Background discussion is provided on JPEG and why it is a strong candidate to become a de facto *standard for interactive digital video, even though MPEG is the ISO standard primarily intended for this purpose. Examples are given of the data rates and typical frame-to-frame variations of JPEG data streams which must be supported by real-time software.*

A simple audio coding algorithm is described, along with a video/audio multiplexed encoding scheme, which was designed both as a practical starting format for JPEG video, and as a vehicle for initial investigation of the more complex MPEG Systems multiplexed encoding format. A description of the hardware testbed designed for our experimental work is given, and this leads to a discussion of how the task of keeping video and audio streams "relatively" synchronized during playback can be managed with minimum burden on the operating system. Finally, the real-time software design which runs on the host Unix workstation is summarized, and key issues for system resource management are highlighted.

1 Introduction

One of the exciting prospects of multimedia systems is that full-motion video has the potential to become "just another data type." To the computer systems community, this usually implies that video will be digitally encoded, so that that it can be manipulated, stored, and transmitted along with other digital data types on standard platforms, storage devices, and networks. It also implies, naturally, that the digital video encoding will be a standard one.

Standard digital video encodings such as D1 and D2 are becoming increasingly common in video production environments. At data rates on the order of 200 Mbits/sec, they provide excellent picture quality (superior to broadcast video, though below HDTV). But such rates simply overwhelm the storage and interconnect capacities of today's networked computing systems, or at best make digital video too costly for widespread application.

Modern compression algorithms can reduce the data rate by one to two orders of magnitude, and current VLSI technology can implement these algorithms as single- (or few-) chip real-time video codecs. These algorithms and silicon implementations become the enabling technology to make digital video commonplace in networked computing systems.

Use of video compression for desktop multimedia applications is not new. A proprietary video compression algorithm is used within Intel's DVI multimedia system for PCs, and this system has been commercially available for some time [7,8]. However, the strong demand by the computing industry's customer base for open, interoperable systems means standard interfaces and data encodings are required. This demand has contributed substantially to the strong momentum of JPEG and MPEG, the emerging ISO still and motion image compression standards intended for use in computing environments.

Although JPEG and MPEG are not yet fully approved ISO standards, the algorithmic decisions have been (apparently) firm within their committees for several months. A number of silicon vendors have either built early versions of these codec chips or have announced their intentions to do so.

If a system designer believes these standards will become an accepted part of the future multimedia computing infrastructure, he or she can use these early chips today to gain experience with workstation system issues. In particular, there are issues about which hardware architectures best support a video codec and its associated compressed and uncompressed data streams, and how software designs can best utilize operating system services to provide robust, real-time multimedia toolkits to the application developer.

Some contend that an operating system with true real-time capabilities is required to provide robust real-time multimedia performance. To test this proposition, and to understand other system resource requirements as well, we have designed prototype hardware using available silicon for real-time video and audio compression, and prototype software running under a typical UNIX operating system (without real-time extensions). The software controls the video and audio devices and performs real-time I/O, buffer management, and synchronization.

Key issues include what are the required buffer sizes and management policies to maintain real-time flow of data to the video screen and audio speaker, given typical worst-case interrupt and scheduling latencies. At press time, we were still exorcising some special brands of multimedia bugs from our device driver, so our experimental results will be reported at a later date.

2. JPEG-as-video in light of MPEG

We begin with what may look like a digression on standards, but is in fact an important "design consideration" for multimedia within the computing industry at large.

JPEG and MPEG are the informal names for the sister ISO/IEC committees designated JTC1/SC2/WG10 and WG11, respectively. JPEG's charter is to produce an international standard for compression of grayscale and color *still* images [2]. To be generally applicable across a broad range of applications, the draft JPEG standard defines a baseline algorithm and a number of extensions. MPEG's video charter is to produce a standard for compression of color *motion* image sequences, i.e., digital video, for use primarily on digital storage media in support of interactive video applications [3]. (A related CCITT standard - commonly called px64 - is intended for video conferencing [4].)

MPEG's first video compression standard will provide approximately VHS TV picture quality at a compressed data rate around 1.1 Mbits/sec. This is low enough to meet CD-ROM I/O rates while leaving around 100 Kbits/sec for audio encoding (discussed below). MPEG enjoys tremendous world-wide participation by the computing, consumer electronics, and telecommunications industries. There is little argument that if what multimedia applications need is VHS-quality video at CD-ROM rates, then MPEG will be the dominant standard.

In spite of MPEG's momentum, a strong case can be made that JPEG too will become a motion image standard - one that will coexist, not compete, with MPEG in the long run. (To be sure, a JPEG digital video standard would be a *de facto* standard. ISO has not intended JPEG for motion image coding, and no proposals within ISO have been made to extend JPEG's charter.) Clearly, one reason JPEG-as-video has received attention is the widely-publicized availability of C-Cube Microsystems' single-chip video-rate JPEG codec, coupled with the industry's "technology push" for digital video.

Beyond this reason, most agree that if desktop video production is to be done digitally, an *intra-frame* coding method like JPEG - where each video frame is encoded independently - is required for video capture, to enable easy editing at any frame boundary. When the editing is completed and the video ready for final form, it can be transcoded to an *inter-frame* encoding like MPEG, to achieve the higher compression needed to realize VHS quality on CDROMs, DATs, etc.

Furthermore, it is clear that JPEG will be the standard color image compressed data type for the computing industry: PostScript, TIFF, ODA/ODIF are in some stage of incorporating JPEG into their document standards, and virtually every major computer vendor has announced a JPEG-based software product. The JPEG algorithm will be implemented in software on every desktop, and as RISC CPUs push ever onward to 100 MIPS and beyond, JPEG video will become possible in software at increasing frame rates and sizes.

One can view experimentation with video-rate JPEG both as a way to gain experience with many of the workstation system hardware and software issues which must be solved to integrate MPEG, and as a tool to further explore the viability of JPEG as a (*de facto*) video standard in its own right.

3. Compression rates and quality

Without getting into algorithm details which are readily available elsewhere, this section provides some intuition for the data rates, perceptual quality, and other characteristics of the compressed video and audio streams used in our prototype.

3.1 JPEG Video

To achieve the data-rate reductions of order 100:1 needed to make video manageable in current systems, data must typically be "thrown out" by reducing the frame size and/or rate prior to applying the actual compression algorithm. For example, NTSC is (usually) digitized to yield 640 pixels by 240 lines per field, at 60 fields/sec. Thus - given a clean initial signal - 640x240x60 is considered "broadcast quality." But neither MPEG nor JPEG can compress this signal to 1-3 Mbit/sec without significantly degrading quality further.

The signal is therefore reduced to 320x240x30 by taking only half the pixels in each line and only half the fields, yielding a pixel rate of 2.3 Mpixels/sec. Such a signal is commonly considered (the higher end of) "VHS quality" prior to compression. JPEG input parameters allow quality and compression to be traded off at encoding time. For TV-resolution images containing moderately-complex visual detail, JPEG typically requires about 0.5 bit/pixel for usable quality, 1.0 bit/pixel for fair-good quality, and 1.5 bit/pixel for excellent quality. This yields 1.7, 2.3, and 4.0 Mbits/sec respectively for the 320x240x30 signal. JPEG is generally considered to require three times the data rate of MPEG for VHS picture quality.

To gain some understanding of the kind of frame-to-frame variations in coded data size expected of JPEG, we captured and compressed footage from the movie "Star Wars," with each frame reduced to (in this case) 360x240 samples. For these tests, we set the JPEG encoding parameters to compress toward the lower end of usable VHS quality, for average bit rates in the range 0.5-0.8 bits/pixel.

We captured and measured 200 frames from each of two pieces of footage, to give insight into the JPEG data variations: (1) the storyline preface, which scrolls as if on an inclined plane into the horizon, selected for its high spatial frequency graphics, and (2) the garbage bin scene, selected for its very busy background. Frame sizes for the two scenes are plotted in figure 1.

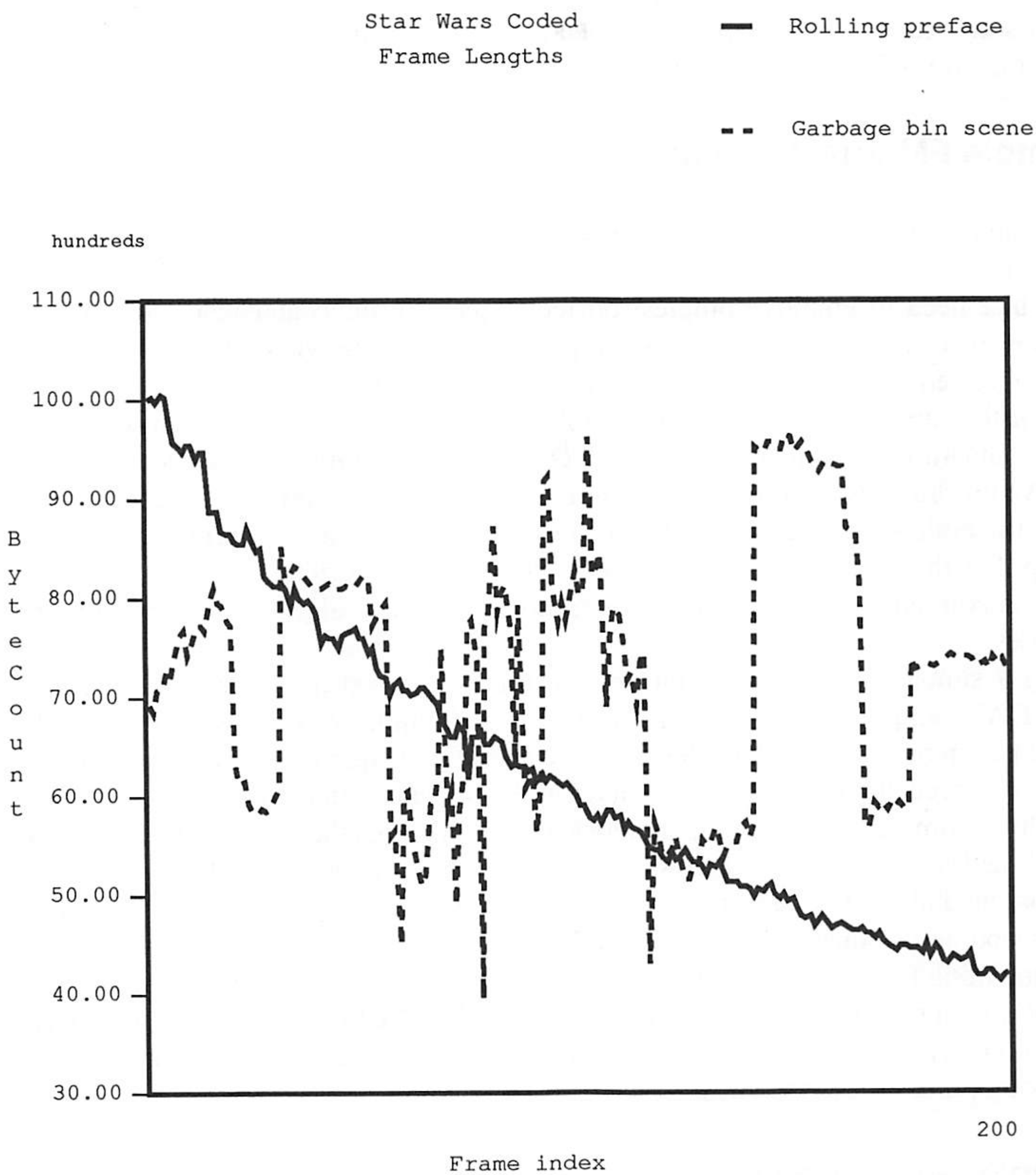


Figure 1 JPEG-coded Star Wars frame lengths

For the preface scene, the number of bytes per frame diminishes with time as the preface fades into the horizon, and less and less text needs to be coded. This is consistent with the observation that fields with text generally need more coded bits than fields without text. For the garbage scene the graph easily reveals several camera changes.

With regard to software design to support JPEG video, a significant point to note is that the variation in size across frames was relatively small: less than 3:1 from the greatest to the smallest number of bits required for any frame. This contrasts with the much greater variation experienced with MPEG coding (6:1 or more) due to differentiation between "Intracoded" and "Bilinearly interpolated" frames in the MPEG video algorithm [3,6]. (In some cases whole frames are coded, whereas in others only differences between frames are coded).

3.2 Simple FM-quality audio

Uncompressed digital audio ranges from 64 Kbit/sec for voice grade up to around 1.4 Mbit/sec for CD stereo. If audio alone is being transmitted or stored, there may be no particular need to employ compression techniques. If audio and video are put together, however, there is motivation to compress audio, for the video data alone - even when compressed - often presses the limits of performance or economy.

One of the subgroups of MPEG, MPEG Audio, has as its goal production of a compression standard for CD-quality audio with compression ranging from 4:1 to 16:1. The MPEG Audio draft algorithm [6] is generally considered too complex - certainly too complex for both stereo channels - to run in real time on a single current-generation DSP chip. For this reason, and because the MPEG Audio algorithm is still in considerable flux in committee, we chose a simple audio compression algorithm for experimental purposes.

Using a simple ADPCM algorithm of the kind described in [9], we compress 48 KHz (R-DAT sampling frequency) audio at 16 bits/sample to 4 bits/sample, for 192 Kbits/sec per stereo channel. This degrades the starting CD quality to the point where it may be considered about FM stereo quality. Unlike MPEG Audio's more complex algorithm, which compresses a contiguous block of samples together as a unit, the simple ADPCM method compresses each sample individually. Though on the surface our sample-oriented algorithm appears not to support random access -- that is, not to be able to start decoding a sequence from points other than the beginning -- a combination of factors stemming from treatment of silences and from audio D/A properties enables random access with only minor transient perturbations. The ADPCM algorithm thus meets our initial quality, functionality, and complexity requirements and serves to simplify the multiplexing process described next.

4 Multiplexed encoding

For real-time playback directly from - or recording directly to - a single digital storage device (magnetic disk, CDROM, DAT, etc.), corresponding audio and video streams generally need to be encoded as a single, multiplexed data stream. This is a practical necessity whenever the aggregate audio and video data rates are a substantial fraction of the device bandwidth, at least for any single-port device with a mechanical seek mechanism: if the data streams were not interleaved together, the seeking (thrashing) would defeat real-time I/O.

If multiplexed data streams are to be interoperable - i.e., playable at least at normal forward speed on different vendors' platforms with proper synchronization - then the multiplexed encoding (as well as the audio and video streams themselves) must be standard. This is the task of MPEG Systems, a third MPEG subgroup operating in conjunction with the Video and Audio subgroups. In keeping with MPEG's overall philosophy, MPEG Systems aims to be as application-independent as is sensible in its multiplexed encoding proposal. This precludes specific encodings optimized for particular devices - such as that described in [11] for optical disks - or for particular industries or applications.

Instead, MPEG Systems has pursued an encoding centered about the video frame. A multimedia program is composed of constituent, or *elementary,* individual streams, e.g., a stream representing one video source or one audio source. A *packet* of data contains material from one of the multiple elementary streams, and multiple packets are multiplexed into a single *pack,* as depicted in figure 2. Packs and packets have internal structure to contain timing information, stream-type identifiers, length fields, and so on.

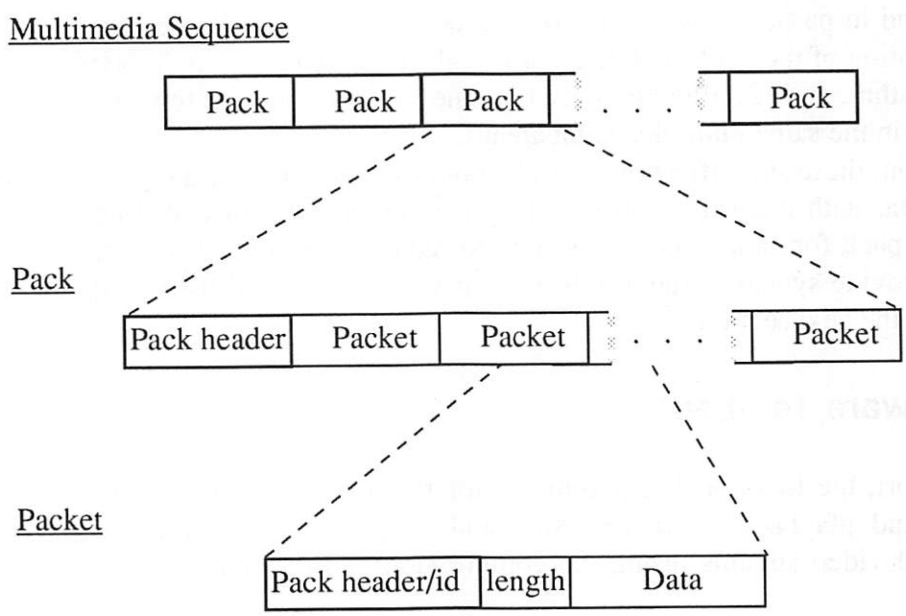


Figure 2 MPEG Systems pack and packet structures

Not conveyed in figure 2 is a pack's temporal structure: playback of pack contents is to be of equal duration (within some tolerance) for all elementary streams present in the pack. The intent is for packs to contain all coded material (audio, video, and other) from 1/30'th or 1/25'th of a second, whenever compressed NTSC or PAL is part of a multimedia program.

The MPEG Systems multiplex encoding is complicated by a number of factors. MPEG audio encodes blocks of contiguous samples together, and individual audio blocks do not generally correspond one-for-one with PAL or NTSC frames (or fields). Moreover, as is explained in [3], the encoded order of MPEG video frames is permuted according to bi-directional interframe encoding relationships, which further complicates association of audio and video time-slices. These and other issues make it difficult, without extensive experimentation, to be reasonably sure that the MPEG Systems proposal will succeed in achieving interoperability across all the device types, applications, and industries it is aiming to satisfy.

One objective of our experimental work is to attempt to validate (or identify the shortcomings of) the MPEG Systems proposal from the viewpoint of the computing industry, and in particular within a workstation environment. We describe here an initial simplification of the MPEG systems proposal by (1) using a sample-oriented audio coding algorithm, and (2) allowing only one one video frame rate (either 25 or 30 Hz but not both) in the same multiplexed bitstream.

Without these simplifications a pack contains only approximately a frame's worth of coded data; with them it contains exactly a frame's worth of coded data. In this latter case, the pack formation serves not only to reduce or eliminate the need for seeks, but also to provide synchronization information without the need for timestamps, as is discussed in the next section.

5 Hardware Testbed

In short, the functional requirements for the testbed are the input and output (i.e., capture and playback, or transmission and reception) of simultaneous synchronized audio and video streams, using the compression and multiplexing methods described above.

For hardware design, a natural starting point is to consider how to combine a JPEG codec chip and a DSP (as audio codec) with a current UNIX workstation. The compressed data rates are very low compared to the workstation I/O-bus's sustainable throughput, but the uncompressed (video) data rates are not. An obvious design approach, then, is to put the compression devices on an I/O bus option card, so that only the compressed video data flows over the bus.

Figure 3 shows a simplified block diagram of the hardware testbed. The workstation platform is the DECstation 5000, equipped with a MIPS R3000 CPU, system RAM (32 MBytes in this case), a SCSI disk, an Ethernet port, provision (not shown) for an FDDI

interface, and a high-resolution (1280x1024) color frame buffer. The dotted-line box encloses the prototype hardware.

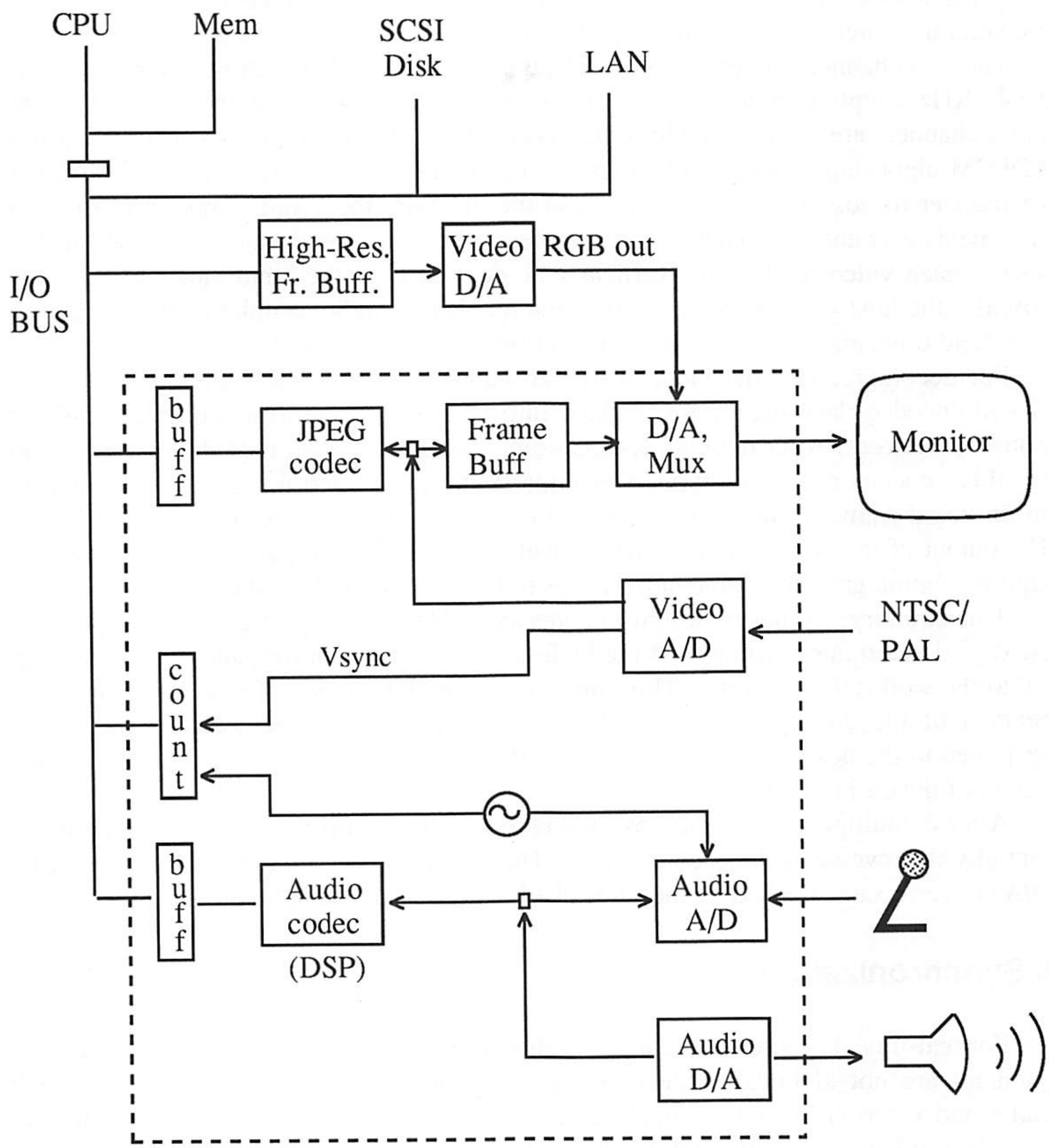


Figure 3 Hardware testbed

Following the prototype's video input path, an external NTSC or PAL video (TV) signal is input and digitized by the popular Philips "TV chip set." The (NTSC) digital video signal is next reduced from 720x240x60 to 360x240x30 by down-sampling circuitry (not shown). This circuitry can also down-sample to other sizes and rates (or not

down-sample at all), to act as a second kind of quality/rate control in addition to the JPEG quantization parameters. The down-sampled digital video signal is then compressed by the C-Cube CL550 JPEG codec chip, under control of the R3000 CPU. The compressed video is buffered for DMA transfers to host memory via the 100 Mbyte/sec (maximum theoretical rate) TurboChannel I/O bus.

The two channels of stereo audio are digitized by CD-quality A/D converters, with the 48 KHz sampling frequency provided by an on-board clock as shown. Both digitized audio channels are input to a Motorola 56001 DSP, which compresses them using the ADPCM algorithm described previously. The compressed data is output and buffered for transfer to host memory. As shown in the diagram, the audio sampling clock also increments a counter for each audio sample, and the counter's value is latched at the start of each video field by the vertical sync derived from the input video signal. This provides the host with an exact count of the number of audio samples occurring during each field time, and is used for synchronization as described below.

For decompression, the video path starts at the host, where chunks of the multiplexed encoded data stream are mapped into system memory from the SCSI disk or from the network. After demultiplexing, compressed video data is DMA-transferred to the JPEG codec on the prototype. Decompressed digital video frames are next input to an on-board frame buffer, with capacity to hold a maximum-size digital video frame. The output of this local frame buffer, as well as the analog output of the workstation's high-resolution graphics frame buffer, goes to D/A and multiplexing circuitry.

This circuitry, similar in design to chips available off-the-shelf today [12], switches the digital video into a window of the high-resolution frame buffer, and outputs the signal to the workstation monitor. This allows the input digital video frame rate to be independent of the display rate, providing a very useful feature for synchronization, as described in the next section. Not shown in the block diagram is scaling circuitry at the output of the on-board frame buffer.

After demultiplexing by the host, the coded audio data follows a path which is essentially the reverse of the encoding path. Though not shown on the diagram, the audio D/A converters are clocked by the same clock used for A/D sampling.

6 Synchronization

For real-time data streams, synchronization problems arise when encoding clock frequencies are not identical to decoding clock frequencies. For applications in which audio and video are captured together and then played back together at an arbitrarily later time, it is usually sufficient that the audio and video streams be only *relatively* synchronized. (This is not sufficient if for some reason the playback time must equal the record time within some tight tolerance.) Events temporally related during capture - lip-sync, start of music at drop of baton, etc. - must maintain the same relationship during playback, at least within the limits of human perception.

The synchronization problem becomes less relative in "live" communications situations, in which data streams are encoded, transmitted, and decoded in real time, without intermediate storage. Either encoding and decoding clocks must be synchronized by one method or another, or data must be discarded from or padded into the stream to

make up the difference. In the following, we focus our discussion on relative synchronization, and we show how this task can be managed using the prototype hardware and the simple multiplexed encoding scheme described previously.

On our prototype hardware, the video and audio encoding clocks are free-running, i.e., are not mutually synchronized. The audio sampling clock is on-board the prototype, but (for capture) the video clock is extracted from the external video source. It is the external video clock which dictates the size of each pack in the stream. The vertical sync signal strobes the audio sample counter, which the host uses to determine how many coded audio samples to interleave with each coded video frame. Note that because audio sample rates do not divide video frame rates, the number of audio samples in a pack varies, even with clocks running at nominal frequencies.

In this way, each pack implicitly contains the necessary relative time-stamp information. The critical timing information is captured on the I/O board, without any host CPU latency entering the equation. The host need only ensure that the input buffers, the audio sample counter, and the devices are serviced once per video frame.

Whereas for capture one can say that the video signal is master in some sense, these roles are clearly reversed for playback, in which our strategy is to ensure that the audio stream plays without adjustment or interruption of any kind. (Omitting or adding audio samples can cause severely unacceptable audio distortion, and attempting to phase-lock the audio clock to the video is problematical for a number of reasons.) Because the frequency of the audio playback clock will generally differ slightly from that of the audio capture clock, this strategy means that the audio playback time will not equal the audio capture time. But with audio crystal oscillators typically accurate to 100 parts/million, at worst this means capture and playback times will differ by about one second per hour, which is not perceptually significant.

The playback synchronization problem thus reduces to keeping the video synchronized to the audio. This is facilitated by our hardware design, in which each decoded digital video frame is written into the local frame buffer on-board the prototype, but is read out asynchronously at a different (much higher) rate to mix it with the workstation's high-resolution graphics video signal. This means that the video can be slaved to the audio: the start of the next decoded video frame is clocked into the on-board frame buffer according to when all the audio samples from the previous pack have been decoded.

One way to control each video frame's start time is to have the host CPU issue this command, based on an interrupt from the audio circuitry signaling that the previous pack of audio samples have completed. In this case, any variation in interrupt latency will be manifest as jitter on the output digital video frame rate. If such jitter degrades motion rendition, another approach is to control video frame start times locally from the on-board audio circuitry. If the former approach proves practical, it will mean that greater flexibility can be given the multimedia system designer, because video and audio control circuitry will not require direct interconnection, but can be controlled through the host.

Whichever implementation approach is used, the foregoing analysis suggests that explicit time-stamp fields are not required in the multiplexed data encoding format in order to keep video and audio streams relatively synchronized in steady-state playback. But even if our experimental results validate this proposition, the need for such time-

stamps will have to be reviewed in light of issues such as management of start-up delays, encoder or decoder processing delays, and complexities of MPEG Video (the permuted ordering of video frames in the stream) and MPEG Audio (the block-of-samples encoding).

7 Testbed Software

7.1 Process Structure

This section describes a digital video cassette recorder (VCR) sample software application being used to validate our model of coprocessing between a Unix host and our prototype hardware. The VCR application is composed of software running in three separate processes - a parent process, a child process, and a driver - depicted by dotted boxes in figure 4. The parent process acts as the entry point and spawns a child process to format coded audio and video data and to ferry the data between disk and prototype. Our focus is on the low-level mechanisms supporting sustained real-time playback and capture of audio and video, and therefore on the child and driver processes.

The primary decoding operations - reading an interleaved stream from disk, demultiplexing the coded video and audio, and transferring these data to codec hardware - are executed by the child process' modules labeled "Storage," "Multiplexing/Demultiplexing," and "Codec I/O and Control." The primary encoding operations - transferring data from the codec, multiplexing, and writing to disk - are likewise executed by the above three modules.

Module execution follows a bucket-brigade model: one module's output is the next module's input. Modules rely on FIFO buffers resident in global memory for inter-module communications; on any given pass, a module is only engaged if its input FIFO contains material to be processed. These FIFOs are loaded with pointers to coded audio and video to avoid needless copying of potentially hundreds or more kilobytes per second.

By way of example, consider inter-module communications on playback. The "Storage" module reads interleaved audio and video from disk and places the data in memory; the module next loads the demultiplexing module's input FIFO with pointers to the newly read data. Thereafter, the coded data are not collectively accessed until the final transfer from memory to audio/video codec hardware. Although the multiplexed stream is scanned for pack and packet start codes as part of demultiplexing, the scan operation is fast compared to a copy operation. (Only a minute fraction of all bytes need be probed since packet length fields can be used to skip coded video and audio).

Low-level control of the hardware occurs in the module labeled "Codec I/O and control." Control operations include initialization, downloading of compression attributes, and issuance of start, stop, and pause instructions. I/O operations include programmed I/O of coded audio and DMA I/O of coded video. Buffer management operations, such as the freeing of dynamic memory once DMA transfers are completed, take place here. Finally, pack formation operations not found in the driver reside here.

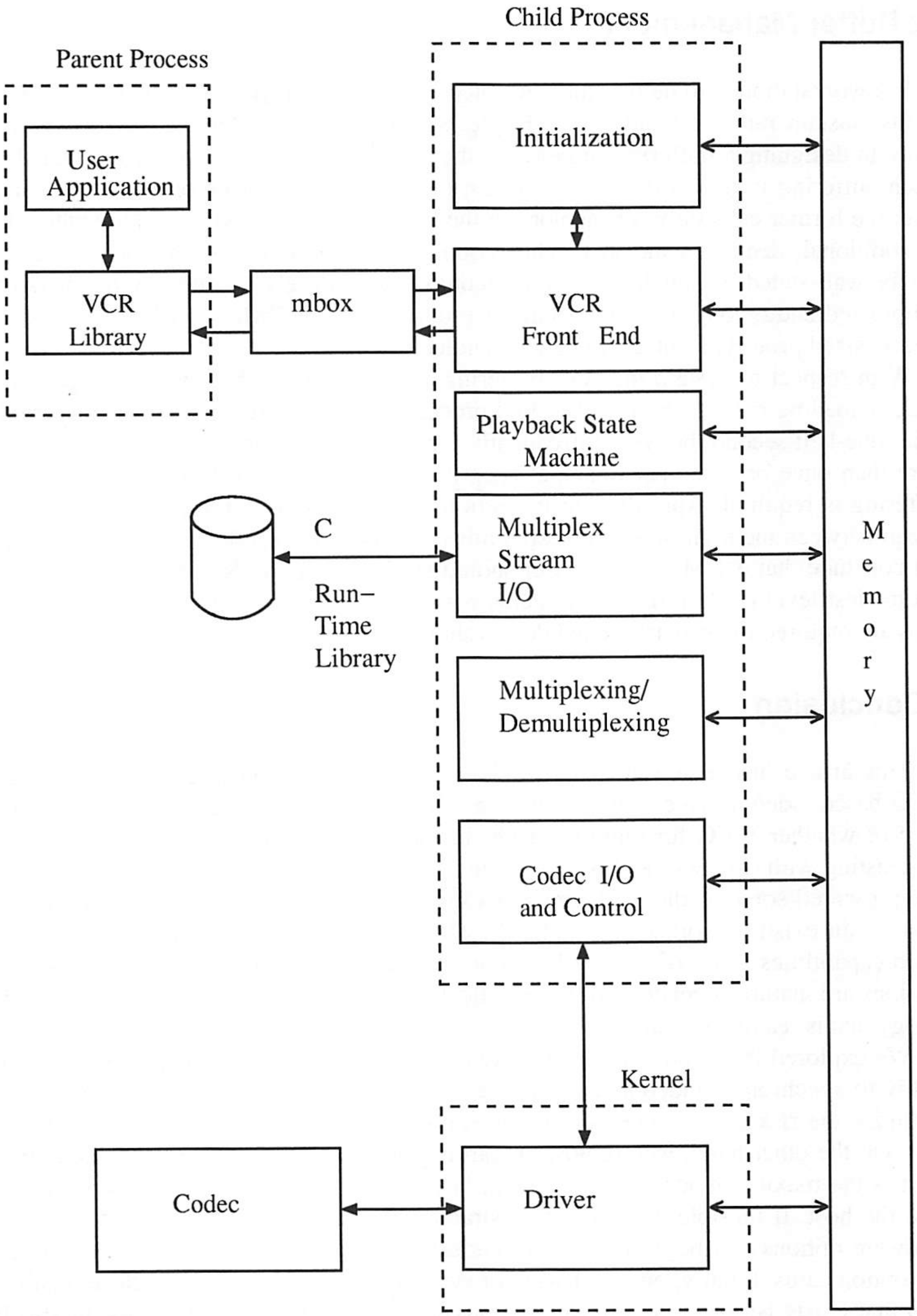


Figure 4 Testbed software structure

7.2 Buffer Management

If a workstation is to deliver uninterrupted capture and playback of audio and video, it must sustain rates of hundreds of Kbytes/sec for video and tens of Kbytes/sec for audio. In designing a buffering strategy for these rates it is important to differentiate between buffering in the workstation's system memory and buffering on prototype hardware; the former calls only for memory in the base workstation, whereas the latter calls for additional, dedicated memory. This, coupled with the observation that Unix tasks may be wait-stated for hundreds of milliseconds, led us to rely on interrupt handlers for compressed audio copy operations and for programming of DMA transfers. We consequently sized prototype buffers to match handler latencies.

With respect to system memory buffering, it is CPU task scheduling that governs. Under a loading model whereby the child process is regularly scheduled out for periods under one-half second, but very infrequently for periods over one second (say, no more than once or twice per hour), at least 0.5 - 1 second's worth of system memory buffering is required. Appealing to to section 3.1, we note that there is roughly a 3:1 spread between the highest and lowest number of bits used to code an individual frame, and conclude that a 1 Megabyte buffer should suffice for a 300 Kbyte/sec stream. This is a modest level of buffering for systems with ten or more megabytes of memory, but steps are required to keep associated delays short.

8 Conclusion

This article has reviewed many of the issues we have considered for integrating JPEG-based video into a current-generation UNIX workstation. We discussed the broad issue of whether JPEG for motion images has a long-term role as a *de facto* standard (co-existing with MPEG), and presented an argument that it does have such a role. We also presented some of the hardware and software design issues one must confront to enhance an existing workstation - like the DECstation 5000 - with digital video and audio capabilities using off-the-shelf silicon in a hardware-option design. The two discussions are naturally related, for the practical reason that it is JPEG-based video technology that is readily available now.

We explored the important "relative synchronization" issue, and explained how the ability to synchronize video and audio circuitry directly to each other in hardware can minimize the risk to real-time performance, compared to leaving this control with the host. On the other hand, we are now embarking on real-time experiments to determine what is the maximum acceptable interrupt latency and variability if this control is left with the host. If feasible, the latter is desirable because it means that video and audio hardware options can be designed independently of one another, e.g., on separate I/O bus option cards. Finally, our discussion of synchronization indicated the close relationship that exists between the information provided (whether implicitly or explicitly) in the audio/video multiplexed encoding scheme, and the actual real-time mechanisms which transfer data, manage buffers, and maintain synchronization.

9 Acknowledgments

The authors would like to acknowledge the efforts of their colleagues, without whom this work would not have been possible. In particular, they would like to recognize Timothy Hellman, who did the initial conceptual work and hardware design for the video and audio codec and host interface, including understanding the synchronization issues. Ken Correll designed the dual frame-buffer video multiplexing circuitry; Paramvir Bahl designed and implemented the ULTRIX device drivers; Davis Pan designed and implemented the audio compression algorithm.

10 References

[1] Digital Compression and Coding of Continuous-Tone Still Images, Part 1, Requirements and Guidelines. *ISO/IEC JTC1 Committee Draft 10918-1*, February 1991.

[2] Wallace, G. K., The JPEG Still Picture Compression Standard. *Communications of the ACM*, April 1991.

[3] Le Gall, Didier, MPEG: A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, April 1991

[4] Liou, Ming, Overview of the px64 kbit/s Video Coding Standard, *Communications of the ACM*, April 1991

[5] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Second Edition. Addison Wesley Publishing Company, Inc, Menlo Park, 1990.

[6] Digital Compression of Motion Images for Digital Storage Media, Parts 1-3, *ISO/IEC JTC1 Committee Draft 11172*, Spring 1991.

[7] Ripley, G. D., "DVI - A Digital Multimedia Technology," *Communications of the ACM*, July 1989, pp. 811-822.

[8] Luther, A. C., *Digital Video in the PC Environment*, N.Y.: Intertext Publications, McGraw-Hill Company, 1989

[9] Pohlmann, K. C., *Principles of Digital Audio, Second Edition,* Indianapolis: Howard W. Sams & Company, 1989.

[10] Georganas, N. D., Yatsuboshi, R., and Naffah N. (eds.), "Multimedia Communications," *IEEE Journal on Selected Areas in Communications*, Vol 8, No. 3, April 1990.

[11] Yu, C., et. al., "Efficient Placement of Audio Data on Optical Disks for Real-Time Applications," *Communications of the ACM*, July 1989, pp. 862-871.

[12] Chips and Technologies, Inc., *82C9001 PC Video - Video Windowing Controller - Advance Product Information, Rev. 1.2*, January 1991, San Jose, CA.

Bernard I. Szabo is currently a member of Multimedia Engineering Advanced Development at Digital Equipment Corporation. Since 1988 he has participated in the MPEG Video and MPEG Systems committees (ISO/IEC JTC1/SC2/WG11), chairing two ad hoc subgroups within these. He holds EE degrees (PhD 1988, SM and SB 1982), from the Massachusetts Institute of Technology. His research interests include integration of real-time media into the workstation environment, image coding, and image sequence motion estimation.

Gregory K. Wallace is currently Manager of Multimedia Engineering Advanced Development at Digital Equipment Corporation. Since 1988 he has served as Chair of the JPEG committee (ISO/IEC JTC1/SC2/WG10). For the past five years at DEC, he has worked on efficient software and hardware implementations of image compression and processing algorithms for incorporation in general-purpose computing systems. He received the BSEE and MSEE from Stanford University in 1977 and 1979. His current research interests are the integration of robust real-time multimedia capabilities into networked computing systems.

Shared Video under UNIX

Paul G. Milazzo
Bolt Beranek and Newman Inc.
milazzo@bbn.com

Abstract

The BBN Video Information Server, constructed by the Multimedia Systems Group, serves several video workstations in nearby offices. The server and workstations are both based on a combination of UNIX® workstations and Amiga personal computers, and communicate using a special video-request protocol layered atop SUNRPC.

The server supports a wide variety of video applications, including interactive video browsers and editors, programs that use moving images as user–interface objects, and programs that automatically index video that has closed captions.

Our experience suggests improvements both in UNIX and in workstation hardware. Native support for bandwidth-reservation protocols and multicasting, and synchronization primitives for character I/O devices would be invaluable, as would better clocks.

1. Interacting with Video

Of the new media entering the computer domain, video is particularly exciting because moving images are a powerful interface to the human mind. From a moving image you can instantly grasp the evolution of a system, be it a historical record, a numerical simulation, or the expressions on your listener's face.

The last example implies a bidirectional, inherently interactive use of video; with a computer, however, even stored video should be interactive. Furthermore, a true multimedia platform would extend support of video information to collaborative applications. Thus, besides merely displaying motion video, a useful video workstation must:

- Provide an input channel—such as an extension of the workstation's pointing device—through which the user can interact with the displayed images.
- Provide a way to combine those images with graphical user-interface objects specified by the application software.
- Support multimedia conference software [Crow90a, Crow90b] that creates *shared workspaces*. For example, such software might need to display a particular video sequence simultaneously to each conferee, perhaps even across large distances.

Placing the video directly in a window on the workstation screen best integrates the special input channel with the workstation environment; for example, Athena Muse made early use of special video-window hardware to present multimedia documents on the workstation screen [Lamp88, Levy87]. Nevertheless, a separate video display is sometimes valuable; for example, a teleconference image has more presence there than in a tiny video window.

1.1. Computer Hardware and the Geometry of Video

Motion video ranges from the high-quality images generated in a television studio to the low-resolution, reduced frame-rate images characteristic of teleconferences.

1.1.1. Characteristics of Broadcast-quality Video

Broadcast-quality video is motion imagery that meets the standards for a commercial television broadcast; in the US, it has approximately 29.97 interlaced 525-line frames per second, usually in color. In each frame, 485 lines contain the image; the *Vertical Blanking Interval* (VBI) comprises the remainder [Bens86].

Because broadcast video is an analog signal, there is no fixed number of pixels per line. For a rectangular sampling pattern, however, sampling frequencies that are multiples of the color subcarrier frequency, f_{sc}, yield the best signal-to-noise ratio. A sampling frequency of $3f_{sc}$ is enough to avoid aliasing, but $4f_{sc}$, about 14.3 MHz, is far more commonly used.

The D-2 digital video standard uses $4f_{sc}$ sampling and 8-bit resolution [Paul89]. Assuming that horizontal blanking consumes 16% of each line, there should be 764 samples/line. The overall television image has an aspect ratio of 4:3, so the resulting pixels are not square. In fact, they are $764 \cdot ¾ / 485 \approx 1.18$ times wider than they are high; this fact causes no end of trouble when placing digitized video on square-pixel workstation screens.

Serial transmission of D-2 video requires a bandwidth of $4f_{sc} \cdot 8$ bit ≈ 114.5 Mbit/sec.

1.1.2. Typical Hardware Support for Real-time Video

Displaying motion video makes stringent demands on a typical workstation's ability to handle and synchronize a voluminous stream of information. Thus, almost any marriage of workstations and video requires special hardware.

Several workstation vendors now provide *video-window* hardware, which digitizes an incoming full-motion video signal and displays it in a window. It is infeasible to transmit the digitized video across the bus of a conventional workstation; most video-window hardware either uses a private path to the frame buffer or simply replaces it. A few video-window implementations support *digital optics* effects, which include zooming and replicating portions of the digitized video and mapping it onto arbitrary three-dimensional surfaces.

Real-time compression of high-quality video is just now appearing on UNIX® platforms; most use the CCITT/ISO Joint Photographic Experts Group (JPEG) algorithm [Wall91], which reduces the bandwidth of a digital video signal by a factor of at least 25 without significant damage to the images. The compressed data is well within the bandwidth of workstation busses and disks; nevertheless, the decompression step still requires direct frame-buffer access to display the reconstituted video.

1.1.3. Non-real-time Video

For many applications, the ability to capture and display a single still frame suffices. The hardware required to capture one frame from a continuous analog video stream is relatively inexpensive, and once captured, all subsequent handling can be done in software.

Given a video player capable of stepping through recorded video one frame at a time, one can digitize each frame of a motion-video sequence, storing or transmitting the data for subsequent reassembly. The cost of a video recording device capable of assembling these frames into motion video can easily exceed that of a workstation equipped with real-time video compression hardware; still, this technique is useful for creating still-image collections such as time sequences.

2. How Software Can Use Video

Once integrated into the workstation environment, video is available as another resource that application software can manipulate. Interactive software can passively display video, offer the user interactive control of the video, or use the video itself as an interaction object. Noninteractive software can operate autonomously to collect and index incoming video streams.

2.1. Classes of Video-based Interactions

In *passive display,* the application software simply obtains and displays video information. Such operation is appropriate for inherently uncontrollable video sequences; these include externally-supplied video streams such as broadcast television, and sequences consisting of a single frame. The display is only ''passive'' in the sense that the viewer cannot alter the presentation within a particular sequence; users can still interactively choose the sequence: perhaps by changing the channel or selecting a particular still frame.

Interactive control software typically allows the user to view a particular video sequence in many ways, forwards and backwards at varying speeds. Such software implements some or all of the functions of a traditional video editing console; some versions also offer simultaneous access to sequences stored on a variety of physical media.

Perhaps the most interesting video applications involve *geometry-based interactions,* in which the video images themselves form part of a user-interface object. Typically, the software interprets the user's gestures atop various areas of still frames or motion sequences as requests for some action; for this interpretation, it requires detailed information about the contents of each frame, and some way to determine which frame was visible at the time of a particular gesture.

Sometimes, collecting video information requires operations that are very slow or must begin at unfortunate hours of the day. *Noninteractive* video software can perform many functions—such as digitizing long video sequences or scanning for keywords in closed-captioning information—that users would find extremely tedious.

2.2. Handles for Application Software

A suitable platform for video-based applications must provide, at some level, interfaces to the analog video world. These interfaces should include:

- a programming model for video devices: players, tuners, switches, and the like,
- support for synchronizing displayed video with other input/output operations, and
- a method for extracting embedded information, such as time codes or closed captioning, from the video stream.

The synchronization tasks are particularly difficult under UNIX. For example, a program reading characters from a closed-caption decoder might need to know which video frame was displayed when a particular character arrived; unfortunately, character I/O latency is unpredictable.

3. Video Application Software

The Multimedia Systems Group at BBN has developed the *Video Information Server,* an automated video library that serves a local cluster of video-capable workstations. Using a toolkit that provides an object-oriented interface to the resources of the Video Information Server, the group built a variety of video application software that executes on the video workstations:

- Passive-display applications:
  - `tv`, which appears as a small television-set icon, tunes television channels, displays test patterns, and sets up interoffice video conferences.
  - `vlens`, a simple pattern-based catalog browser, retrieves still-frame and motion video sequences from an interdisciplinary collection of video information.
- Interactive-control applications:

- vide is a video ''editor'' with which a user can alter the Video Information Server's catalog as well as create custom views of the collection.
- The BBN/Slate Multimedia Editor [Liso89] can encapsulate references to portions of the Video Information Server's collection and transmit them as electronic mail. Recipients can view these references as a passive display, or invoke vide to alter them.
- dub is a true video editor, with which a user can assemble video sequences onto recordable media such as videotape or write-once videodisc.

- Geometry-based interactions:

  - Xnavigate displays still-frame images of maps and charts stored on videodisc. Mouse gestures atop the displayed map zoom and pan the view and switch to different types of maps of the same area. In areas where the map database is well-populated, the user can begin with a world map and zoom in close enough to see individual buildings.

- Noninteractive applications:

  - autodub records television programs for later viewing. A companion application extracts text from the closed-captioning stream, if available, and adds it to the Video Information Server's catalog. Users currently invoke autodub from at or cron; eventually, a centralized service will combine user's interest profiles with machine-readable television programming information to generate recording schedules.
  - A pair of programs perform unattended frame-range digitization and animation assembly; these operations often run for many hours, usually at night.

All of the interactive programs can run as shared workspaces under the MMConf multimedia conference structure [Crow90b]. Conferees within range of the same Video Information Server share access to the physical video media used in the conference. Given Video Information Servers at distant sites, conferees clustered around those sites would receive their video images from the local server

3.1. tv

The tv application normally appears as a television-set icon on the desktop. Through menus popped up from tv, the user can tune television channels, display test patterns, and start an interoffice video conference. It is table-driven, and uses the Video Information Server's *mount* facility to tune channels. The server models video information sources as *volumes*; in Figure 1, tv requests volume *WNEV,* which the server recognizes as appropriate to mount on a device of type *tuner*.

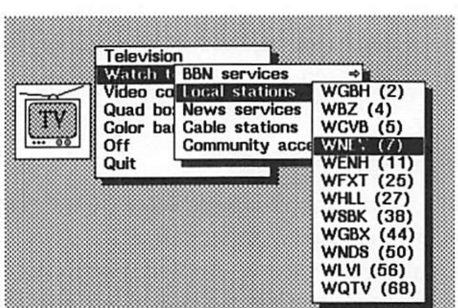


Figure 1: Using tv to tune WNEV.

3.2. vlens

`vlens` is a very simple tool for locating, by regular-expression search, any video clip known to the local Video Information Server. It combines ''card-catalog'' searching with actual retrieval of the desired clips. In Figure 2, the user searched for all video clips about Iraq, then selected a specific entry in the list. `vlens` responded by filling in the area at the bottom with the duration (20 seconds), the name of the volume on which the clip resides, and the full text description. A double-click will play the clip.

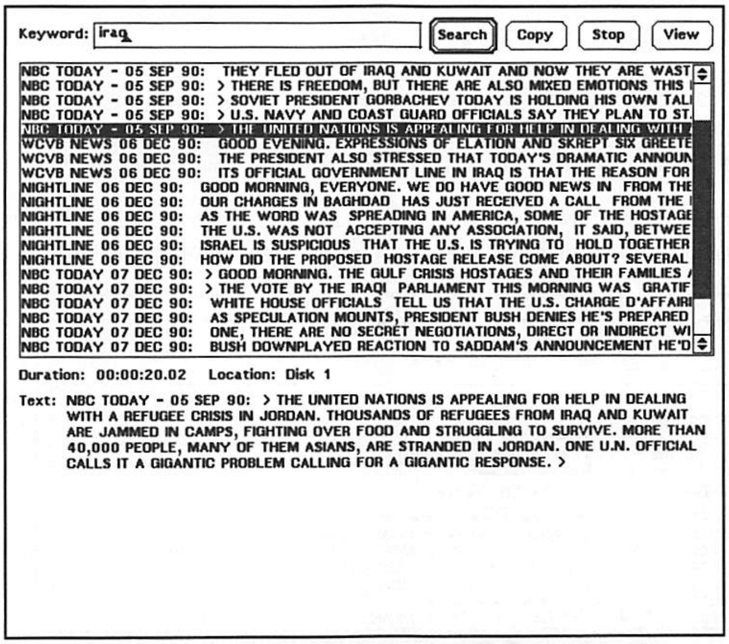


Figure 2: Searching closed-captioning information for news about Iraq

The database that `vlens` searches contains the union of:

- the index information provided with the prerecorded media in our collection,
- information automatically extracted from closed-captioning text, and
- entries created manually.

The particular entries found in Figure 2 are closed captions.

Besides viewing individual clips, the `vlens` user can select multiple entries from the list, copy references to them into a Slate clipboard, and then paste the references into a BBN/Slate document as an *enclosure* (Section 3.4.).

3.3. vide

`Vide`, the video editor, is a general tool for browsing structured or unstructured video media. It operates on simple, generic descriptions called Edit Decision Lists (EDLs). Figure 3 shows a short EDL that includes a still frame and motion sequence from each of two volumes: *Voyager Gallery 1* and *Earth Science 1*.

`Vide` is not a true video editor because it cannot control a recording device. In the conference environment, it is more important to be able, independent of the actual storage media, to present or browse sets of video clips; actual assembly of video productions is better done with existing systems.

```
EDLVersion 2.0
# created by milazzo@pearl on Wed Nov  1 13:32:02 1989
# in    out    speed   scene   volume
45784   45784   1/30    "*Comet Halley; 60-image composite obtained with the Halley
        Multicolour Camera during the European Giotto spacecraft flyby. Sun is to left and
        North is up. Range was 600km from nucleus on 3/14/86. Resolution from 800m at lower
        right to 100m a t the foot of the jet. Surface has low (4%) reflectivity suggesting
        a covering of a dark, porous mantle. Surface temperature is 100F to 50F in sunlight
        and core temperature is -400F to -300F. Image processing by Ball Aerospace. Image
        and footage following courtesy Alan Delamere and Harold Reitsema." "Voyager
        Gallery 1"
45785   46443   1       "HALLEY/GIOTTO 3-HOUR FAR ENCOUNTER DATA ANIMATION"
1751    1751    1/30    "GLACIAL PROCESSES: Sheridan Glacier; E of Cordova, AK"     "Earth
        Science 1"
32189   33685   1       "MOVIE:  Tornadoes - Animation of tornado formation and actual
        tornadoes"
```

Figure 3: An Edit Decision List: each entry specifies the volume, frames, speed, and label

Figure 4 shows the `vide` user interface. The *Editing Control* section manipulates and plays EDLs in part or whole; the remaining sections allow direct control of video devices.

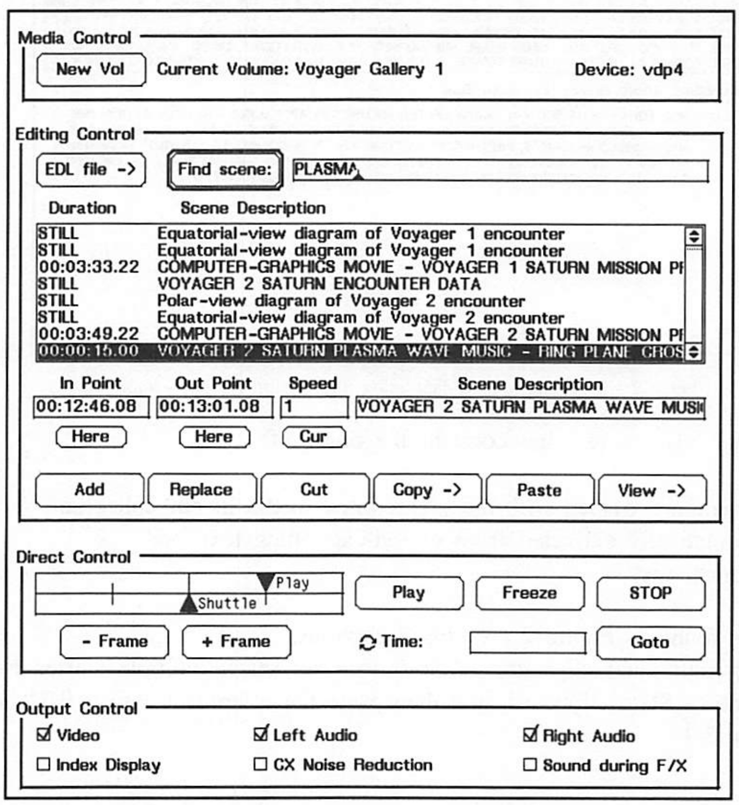


Figure 4: Examining a videodisc with `vide`

The Editing Control section reads and writes EDL files, and searches for patterns in the Scene Description field. It allows modifying and previewing ranges of the current EDL, and can write portions of it as a BBN/Slate enclosure. The user can enter edit points manually or take them directly from the current video device. Edit points can appear either in standard SMPTE (Society of Motion Picture and Television Engineers) notation, as shown in Figure 4, or as absolute frame numbers.

The *Media Control* section provides a way to select, from a list, any volume in the Video Information Server's collection. Volumes can be videodiscs, tapes, files containing digitized video, cameras, or television channels.

The *Direct Control* section contains the variable-speed play, jog and shuttle, and random access controls. The speed controls—Play and Shuttle—are exponential, with a range of 1/1000× to 100× normal speed in either direction; each video device approximates the chosen speed as best it can. The Goto function moves to a specific time, absolute or relative frame, or chapter.

Finally, the *Output Control* section enables and disables the video and audio outputs and device-specific video overlays and audio processing.

3.4. Video Enclosures

A BBN/Slate document can contain *enclosures,* uninterpreted data objects plus the names of external programs that implement a set of standard methods such as *edit* and *execute* [BBN90a]. Several EDL-based video applications, including `vlens` and `vide`, can create enclosures containing EDL fragments, and write them to the BBN/Slate clipboard [BBN90b].

In Figure 5, the author used `vlens` to select a collection of clips, copied it to the clipboard, pasted it into a BBN/Slate document, and mailed it to a colleague. The recipient invokes the enclosure's *execute* method to view the collection.

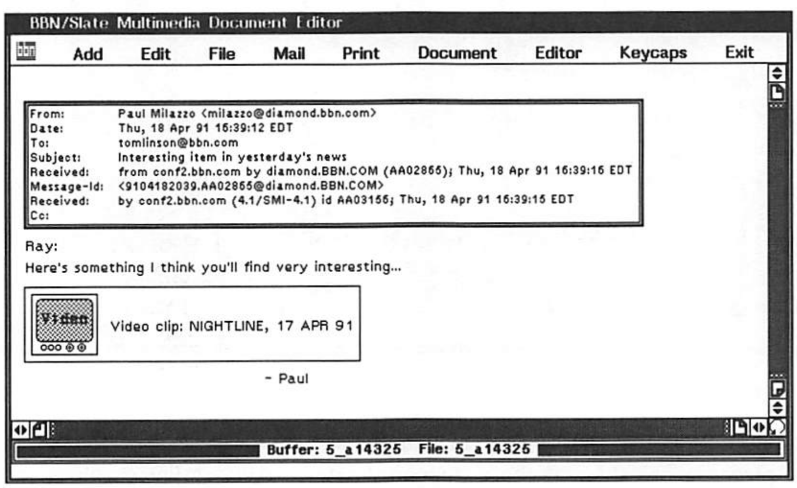


Figure 5: BBN/Slate electronic mail containing a reference to video stored on the Video Information Server

3.5. `Xnavigate`

`Xnavigate` is a specialized application for collaborative map-browsing. It uses a cartographic database to locate still images—photographs of paper maps—on videodiscs supplied by the Defense Mapping Agency (DMA) [DMA86]; the Video Information Server loads the required videodisc and displays the images. Graphical overlays added at the user's video workstation create the display of Figure 6.

Clicking the direction arrows moves the field of view in that direction; clicking elsewhere on the display zooms in closer to the selected point. A dedicated mouse button brings up a menu of special functions: zooming in or out without changing the current geocoordinates, adjusting the position of the graphical overlay, and terminating `Xnavigate`. Another dedicated mouse button displays the *telepointer,* a large cursor simultaneously visible at all conference sites [Crow90b]. Finally, `Xnavigate` allows freehand sketching simply by holding down a mouse button and dragging the mouse across the video image.

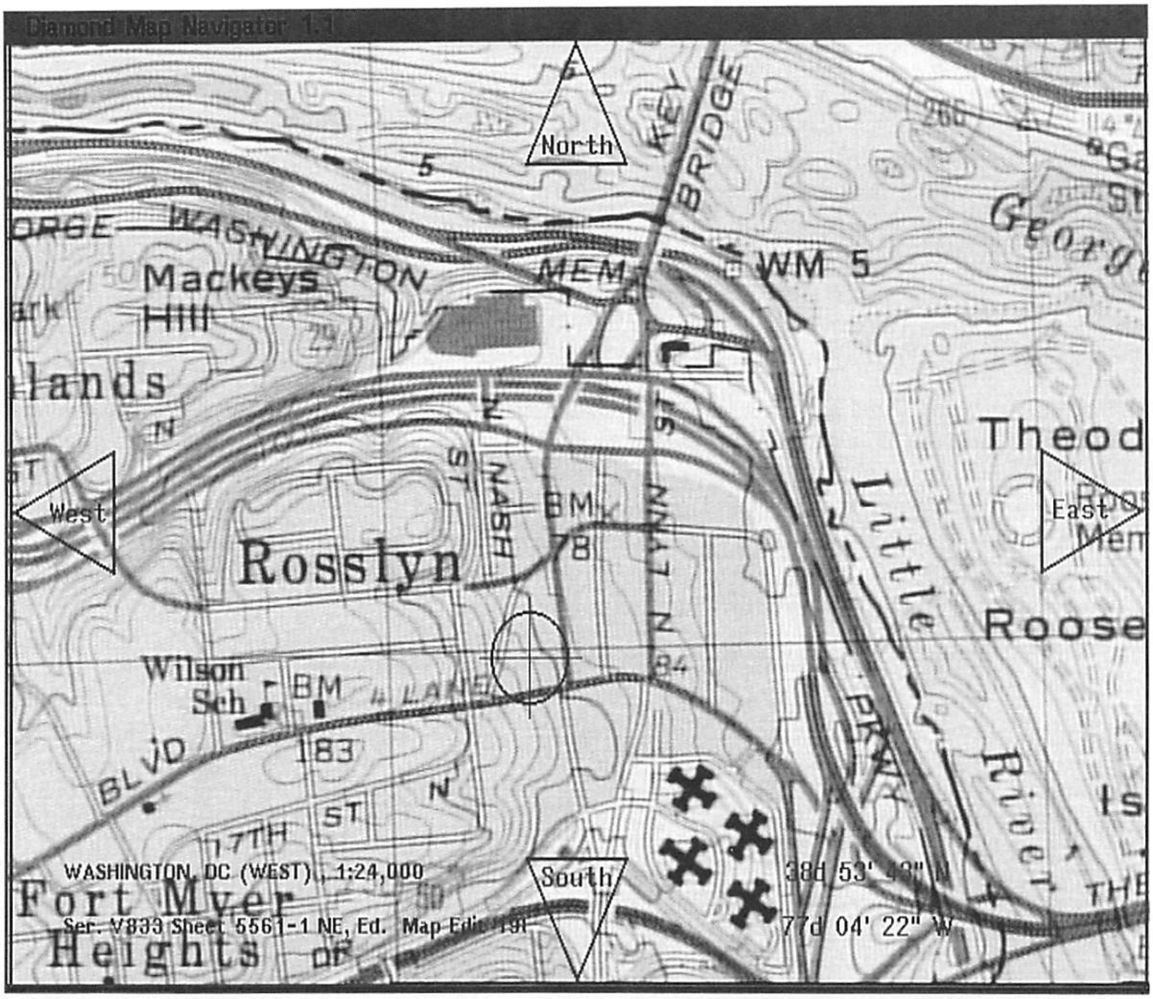


Figure 6: The Xnavigate display: a map overlaid with user-interface graphics

Textual annotations identify the current map and the geocoordinates of the last mouse selection; `Xnavigate` also places a crosshair at those geocoordinates. All overlay graphics appear in magenta because DMA maps almost never use that color.

4. Video Workstations

The video-window hardware available in 1988 was very expensive, both intrinsically and because it required color workstations with several free bus slots. Because the cost of equipping every office with such hardware would have been prohibitive, the author assembled a relatively inexpensive video peripheral that—combined with our existing UNIX machines—formed a useful video workstation.

4.1. The Prototype Video Workstation

The prototype unit, seen in Figure 7, consisted of a standard UNIX workstation connected via Ethernet to a video peripheral that provided a video display, a high-resolution graphical overlay with a cursor, and stereo speakers. The core of the video peripheral is an Amiga 2000 running an X11 server under AmigaDOS; a Mimetics genlock synchronizes the Amiga's CPU clock, and hence generation of the overlay graphics, to the incoming video signal.

Figure 7: 1988 prototype video workstation, with vegetable

An analog network delivers video and audio to each office. Some offices also have a video camera and microphone connected to the network.

Using the genlock to produce a composite video signal containing the overlay graphics would have rendered them illegible. Instead, the author modified the genlock to send the keying signal and Amiga RGB graphics directly to the monitor, a SONY PVM-2530, which can perform the mixing operation in RGB space.

The complete video peripheral cost approximately $5k, perhaps one-fourth the cost of the least-expensive video-window workstation available at the time. Because Ethernet was its only link to the main workstation, it could be added to all of the different UNIX workstations used by the Multimedia Systems Group. Its final advantage was that it usurped no valuable screen space.

Unfortunately, it did usurp much valuable desk space. It was also rather poorly integrated with the native workstation environment, and ill-suited to all-digital video distribution schemes.

4.2. New Video Workstations

In the last year, a variety of workstation manufacturers have introduced cost-effective video-window hardware; unfortunately, it is not yet widely supported by system or application software. Currently, the Multimedia Systems Group uses the RasterOps TC/PIP card in a SUN SPARCstation. The video distribution network is still analog, and stereo sound still requires a separate amplifier and speakers.

Placing the video directly on the workstation screen significantly improves the integration of the video applications, but is not without cost. The video window typically occupies 20% of the area it would on the PVM-2530 screen; thus it loses much of its presence, particularly in teleconference

applications. Nevertheless, it manages to occupy enough screen real-estate to seem forever in the way.

As real-time video compression hardware becomes available on UNIX platforms, the Video Information Server will migrate to an all-digital service; one impediment, however, is the lack of kernel support for multicast stream protocols. Another group at BBN sends digitized voice and video from conventional CODECs over packet-switched networks; they currently use the ST protocol to reserve bandwidth on wide-area networks [Topo90].

5. The Video Information Server

The BBN Video Information Server is actually a distributed service currently provided by five SUN workstations and two Amigas; together they control a collection of video devices. In one case, a SUN and an Amiga are part of both the server and a video workstation in a user's office.

Figure 8: These devices form part of the Video Information Server.

At last count, the Video Information Server included the following devices:

- two VCRs,
- five read-only LaserDisc players,
- one write-once videodisc recorder/player,
- three video frame stores in SUNs,
- two video frame stores in Amigas,
- five computer-controllable tuners,

- a ''quad box'' that combines four video images into one,
- three interconnected video/audio crossbar switches,
- a closed-captioning decoder with an RS-232 output,
- a connection to the BBN packet-switched wide-area video conference facility,
- miscellaneous audio mixers, a sync generator, and other, similar glue.

All of the devices controlled by SUNs have RS-232 interfaces. The Amigas support bus-connected devices such as frame stores, and control a few recalcitrant devices—such as cable-television converters—by using their tone generators to synthesize infrared remote-control signals. Figure 8 shows one of the SUNs, one of the Amigas, and the video devices they control.

5.1. Server Architecture

The Video Information Server provides a device-independent, object-oriented view of its video resources; Figure 9 shows the class hierarchy. Client applications locate video resources by name with local-network broadcasts; they need not know the actual host controlling those resources.

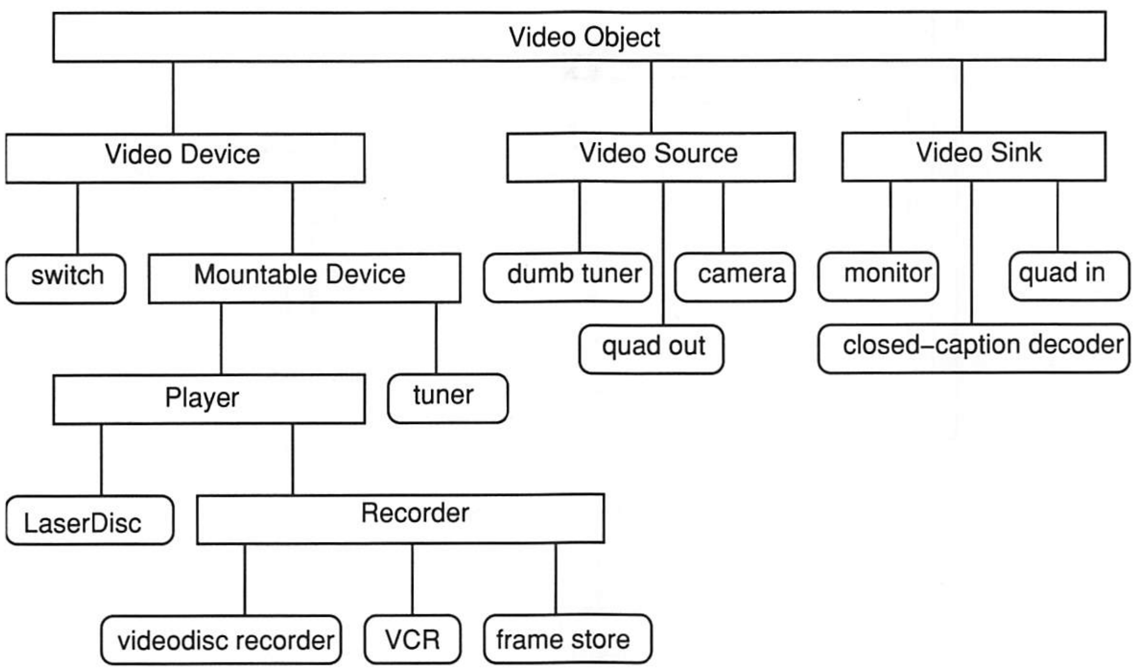


Figure 9: The video class hierarchy

Clients communicate with the server's components using a protocol layered atop SUNRPC [SUN88]. Requests at this level take the form of messages addressed to a particular video object; a typical requests is, ''starting at time *t*, play from frame *m* to frame *n* at speed *s*.'' Most operations that take longer than a few tens of milliseconds run asynchronously; the client can optionally request to wait until all pending operations are complete.

An extensive library of video functions, linked into the client, provides two additional levels of abstraction above the server's native protocol. A typical request at the highest level is, ''perform EDL segment *S* on monitor *M*.''

Each host that forms part of the server runs a single multi-threaded process called `videod` that controls all the host's video resources; `videod` has a standard framework, consisting of a communications module and a thread scheduler, and a collection of video device drivers that

implement the set of methods for the device class. The SUN and Amiga versions use identical frameworks.

The multiple threading is loosely based on continuation semantics [Stoy77]. The dispatcher translates each incoming request into a continuation and adds it to the end of the queue for that device. To wait on a device or timer, the device-specific driver returns a continuation, which the scheduler places on the front of the matching queue. Figure 10 shows `videod` controlling two videodisc players (VDP) and a videodisc recorder (VDR).

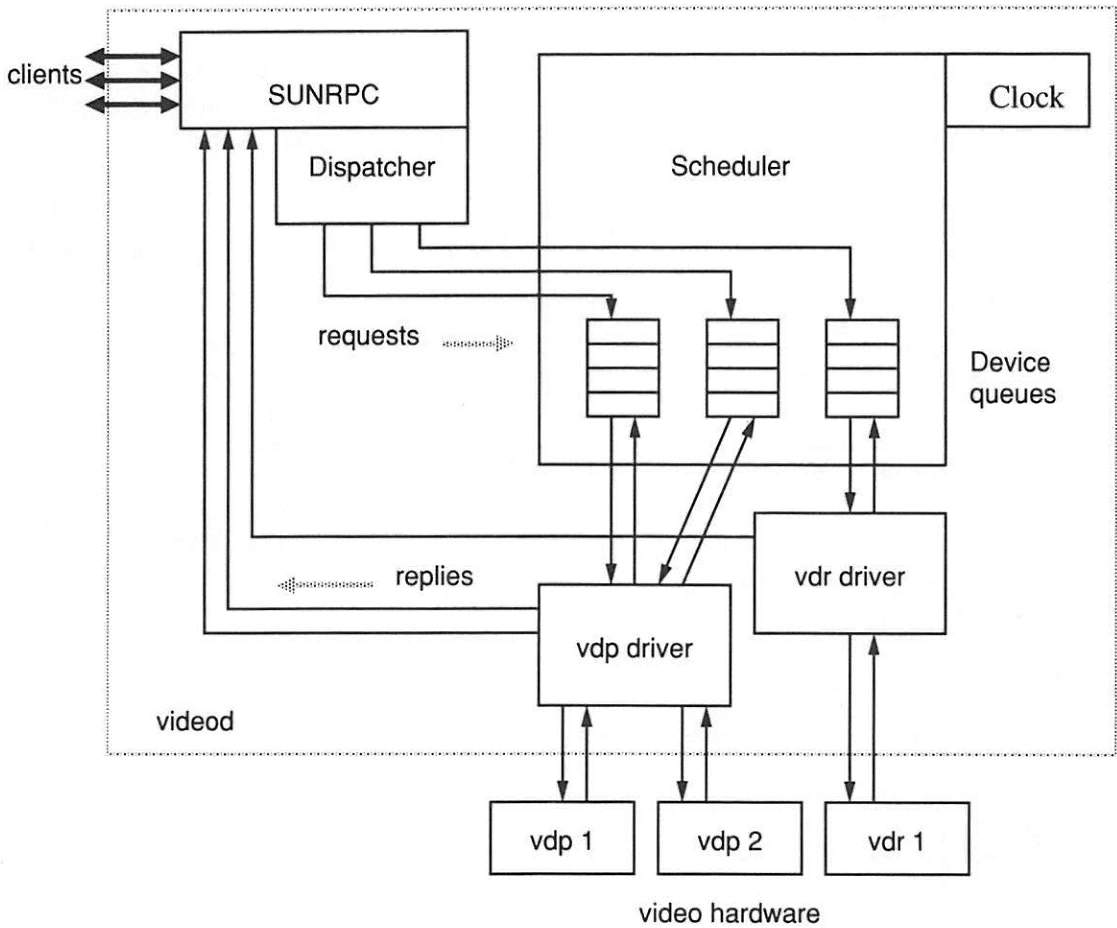


Figure 10: Multiple execution threads in `videod`

5.2. The Catalog

The server has an extensive catalog of its holdings, often indexing tens of thousands of individual frames on a single volume. It extends its collection by recording television programs and augmenting the catalog with closed-captioning text decoded therefrom.

Users who contribute new prerecorded media to the collection must also update the catalog. Videodiscs often come with machine-readable indexes; other recordings sometimes have closed captions. As a last resort, the user can manually create catalog entries.

If file protections permit, a user can modify existing entries in the catalog, adding value by subdividing long sequences or correcting errors in the textual descriptions.

5.3. Replication in Multimedia Conferences

Because the physical switches can connect a single signal source to multiple sinks, replicated applications in a multimedia conference can share access to local devices.

The sharing scheme is simple: when an application running in a conference attempts to mount a volume, only one copy actually performs the mount and receives a valid device name; the other copies are designated *spectators* and receive a specially–tagged device name. For example, one copy of the application attempting to mount the volume *Earth Science 1* might be told it resides on `vdp4`; the others would be told it resides on `vdp4(SPECTATOR)`.

When the other copies attempt to locate the server for `vdp4(SPECTATOR)`, the interface library simply returns a locally–constructed player data object. Any attempt to control this player is quietly ignored; all requests for player status first rendezvous with the copy actually controlling the player, then return the information thus obtained. Each copy can function as though it controlled the player; Figure 11 illustrates this approach.

The sharing scheme only intercepts operations on mountable devices. Other requests, such as connecting the spectator device to the local monitor, must go through, lest the user see no picture.

The current implementation only allows one conferee to be the actual device owner; this technique will fail for conferences involving multiple servers. In the next version, each server will maintain a local registry of ongoing conferences; the first client to register membership in a particular conference will own all that conference's video devices.

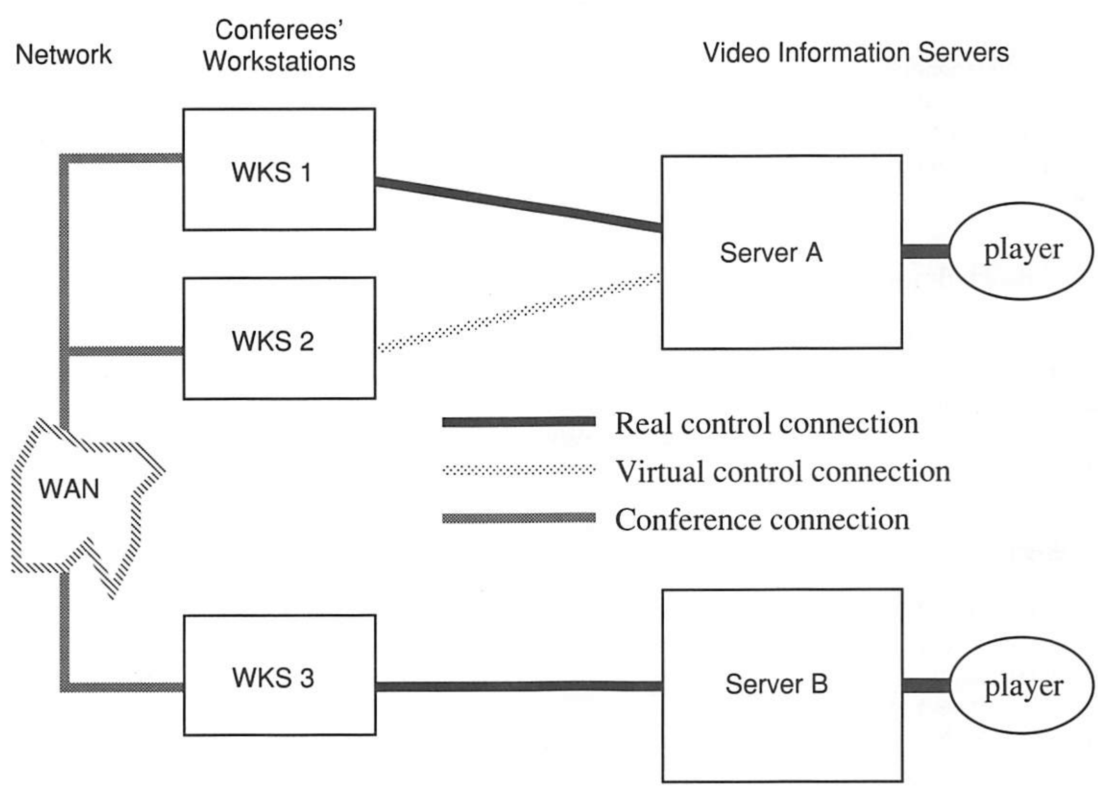


Figure 11: Sharing access to a volume during a conference

5.4. Some Useful UNIX Enhancements

The most burdensome obstacle to implementing video software under UNIX is the unpredictability of character I/O latency. Because most industrial video devices today use RS-232 interfaces,

unpredictable latency makes it difficult to know when to send a device command, and almost impossible to know when an external event occurred.

For example, a program reading closed-captioning characters needs to know which frame number to associate with each character. It might find the frame by reading the associated SMPTE time code from another serial port, by querying the player, or—as a last resort—by estimating based on the current time. Unfortunately, in every case it needs to know the time the character actually appeared at the serial port, not the time the program read it.

Device-driver support to time-stamp each incoming character, and a stream or line discipline that delivered time-stamped characters, would address the most pressing needs. Unpredictable output latency is less tractable, but perhaps the kernel could keep an estimate of the current latency.

Another hurdle is the general lack of adequate clocks. The video protocol allows the client to specify the time at which some operation should execute; for example, the client can queue a number of separate requests marked for simultaneous execution sometime in the future. In order to avoid errors introduced by network latency, the time stamps are absolute, not relative. The hosts use the Network Time Protocol (NTP) to synchronize their clocks [Mill91], but the low clock resolution and lost interrupts common on many workstations make precise timekeeping impossible.

5.5. Similar Systems

Galatea [Appl90], part of the MIT Media Laboratory's Electronic Light Table project, provides a device control model and programming interface much like that of the Video Information Server. Galatea emphasizes reliability and fairness in serving clients; it does not appear to offer shared device access for collaborative work.

The Integrated Media Architecture Laboratory (IMAL) at Bell Communications Research was intended to emulate future telecommunications networks and customer premise equipment. It featured highly-integrated multimedia workstations with access to a variety of video and audio storage devices [Ludw87, Ludw89].

6. Acknowledgments

My thanks to Ray Tomlinson, who helped design the system and implemented the server framework and many of the drivers, and to Harry Forsdick and Terry Crowley, who contributed many ideas and actually wanted video workstations in their offices!

This work was sponsored by the Defense Advanced Research Projects Agency (DARPA) under contract N00140-88-C-8006.

7. References

[Appl90] Daniel I. Applebaum. "The Galatea Network Video Device Control System." MIT Media Laboratory, Cambridge, MA, 1990.

[BBN90a] BBN Software Products. *BBN/Slate Reference Manual* for Release 1.1. Cambridge, MA, June 1990.

[BBN90b] BBN Software Products. *BBN/Slate System Topics* for Release 1.1. Cambridge, MA, June 1990.

[Bens86] K. Blair Benson. *Television Engineering Handbook.* McGraw Hill, 1986.

[Crow90a] T. Crowley, H. Forsdick, P. Milazzo, R. Tomlinson, and E. Baker. *Research in Real-time Multimedia Communications Applications.* BBN Report No. 7325, May 1990.

[Crow90b] T. Crowley, P. Milazzo, E. Baker, H. Forsdick, and R. Tomlinson. "MMConf: An Infrastructure for Building Shared Multimedia Applications." Proceedings of the Conference on Computer-Supported Cooperative Work. ACM SIGCHI & SIGOIS, October 1990.

[DMA86] Defense Mapping Agency. ''Product Specifications for Mapping, Charting, and Geodesy (MC&G) Video Discs,'' first edition. DMA Hydrographic/Topographic Center, Washington, DC, September 1986.

[Lamp87] David R. Lampe. ''Athena Muse: Hypermedia in action.'' In *Project Athena Visual Courseware.* MIT Project Athena, 1988. Reprinted from *The MIT Report,* February, 1988.

[Levy87] Martin J. Levy. ''Live Digital Video in a Windowing Environment.'' Technical Report, Parallax Graphics, Inc., Santa Clara, CA, 1987.

[Liso89] Herb Lison and Terrence Crowley. ''Sight and Sound.'' *UNIX Review,* October 1898.

[Ludw87] L. Ludwig and D. Dunn. ''Laboratory for Emulation and Study of Integrated and Coordinated Media Communication.'' ACM SIGCOMM, Stowe, VT, August 1987.

[Ludw89] L. Ludwig. ''Real-Time Multi-Media Teleconferencing: Integrating New Technology.'' Groupware Technology Workshop, IFIP Working Group 8.4, Palo Alto, CA, August 1989.

[Mill91] David L. Mills. ''Network Time Protocol (Version 3)—Specification, Implementation, and Analysis.'' February 1991. Available by anonymous FTP from `louie.udel.edu`.

[Paul89] C. Robert Paulson. ''Television Signal Transmission: Another Technology in Transition.'' *SMPTE Journal* 98(3). Society of Motion Picture and Television Engineers, May 1989.

[Stoy77] Joseph E. Stoy, *The Scott–Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[SUN88] SUN Microsystems. ''RPC: Remote Procedure Call Protocol Specification Version 2.'' RFC 1057, June 1988.

[Topo90] Claudio Topolcic, ed. ''Experimental Internet Stream Protocol, Version 2.'' RFC 1190, October 1990.

[Wall91] Gregory K. Wallace. ''The JPEG Still Picture Compression Standard.'' *Communications of the ACM* 34(4). Association for Computing Machinery, April 1991.

Paul Milazzo is a member of the Multimedia Systems Group at Bolt Beranek and Newman Inc., where his primary research interests are real-time multimedia collaboration systems and document representations. His personal interests include computer music and book design. He has a BS in Electrical Engineering and an MS in Computer Science from Rice University.

*UNIX is a registered trademark of UNIX System Laboratories.

Compressed Executables: an Exercise in Thinking Small

Mark Taunton

Acorn Computers Ltd
Cambridge Technopark
645 Newmarket Road, Cambridge
U.K. CB5 8PB

mark@acorn.co.uk

Abstract

In the project described in this paper, a large body of software – including a port of 4.3 BSD, Sun NFS and the X Window System – has been made to fit onto small personal computers with comparatively tiny discs and an unusual memory management architecture. The problem of space is exacerbated by the relatively low code density of the RISC processor used, and an enormous hardware-imposed page size might well be expected to make matters worse. Part of the solution is an implementation of shared libraries. In addition, considerable savings, of both disc space and other resources, are obtained by applying a specialised data compression technique to demand-paged executable files. This is done in such a way that the compressed programs are still runnable. In the process the huge page size is turned to advantage by utilising a standard feature of the UNIX file system. Some additions to the kernel, a few extra utilities, and some minor changes in the standard toolkit support the scheme.

We briefly explain the background to this work, and outline the shared library mechanism. The remainder of the paper focuses on the concept of compressing executable code, examining its advantages and its limitations. We suggest possibilities for further improvement of the current implementation, and also some ways in which the technique might be exploited in a broader context.

1. Introduction

Acorn is a small British computer manufacturer. In 1985 we designed the ARM processor, using RISC principles. The following year, three more chips were designed which together with ARM form the core of a low cost 32-bit computer. Acorn began selling complete personal computer systems based on this technology in 1987. Although UNIX did not figure prominently as a target operating system in the hardware design phase – Acorn has traditionally developed its own proprietary operating systems – sufficient hardware functionality is present to support it.

In parallel with proprietary developments, work started to port 4.3 BSD and Sun NFS 3.2, initially to ARM development systems with 8 Mb of memory and around 70 Mbytes of local disc; a

VAX provided the main working filespace via NFS. Later, X11 was ported and it, along with other items, became a standard part of the system. By the time plans were laid for the initial commercial UNIX product, the total volume of the software had become distinctly large. Others have noted and attacked the software bloat afflicting UNIX [For90,Swa90], but there is no easy solution if commercial acceptance is desired.

By contrast, the target hardware for Acorn's first UNIX system product (the R140, launched in early 1989) was very modest, specifically to minimise system cost. Memory is limited to 4 Mb, and the hard disc holds 47 Mb. Of the available disc space, around 4 Mb is allocated to the proprietary ROM-based operating system (called RISC OS[1]) and at least 8 Mb is required for a swap partition. For RISC *i*X (our UNIX port) we were obliged to shoehorn the nominated software into the remaining space as best we could. Some ingenuity was therefore required.

Disc space was saved by applying two separate techniques: shared libraries, and compression of binary executable files. The former is by now well known and documented: the implementation is summarised below (§3). This paper discusses primarily the latter technique.

2. Hardware constraints

Besides the obvious problem of coping with such a small disc, the memory management hardware presented some particular difficulties. It is based upon a very simple and inexpensive design in which all page mappings for each 4 Mb section of physical memory are programmed into a CAM on the memory controller (MEMC) chip. Having all mappings on-chip keeps the speed up but because each CAM cell is relatively large the number of entries is limited to 128. In consequence, with one controller for each 4 Mb the individual page size comes out at 32 Kb. If more MEMCs are used, each controlling less memory, the page size goes down (to a minimum of 4 Kb, for 512 Kb/MEMC). However system cost then increases, so such an approach was not pursued.

2.1. Software difficulties

The consequences of a 32 Kb page size are unpleasant in many respects. The conventional UNIX process management scheme requires at least 4 physical pages (one each for u-page, text, data and stack) to be allocated to a process for it to be runnable. Thus a minimum of 128 Kb is consumed by each loaded process (ignoring the issue of text sharing) regardless of how much work it is actually doing.

Besides the impact on physical memory, disc space is rapidly consumed by executable programs. In RISC *i*X the default format for executables is ZMAGIC, i.e. demand paged, with page alignment on all text and data pages in the prototype file. Aligning page images on page-sized boundaries within the file means that the page-in code in the kernel has no difficulty in calculating the file offset for the start of a particular page's data. The file header, which contains the program section sizes, magic number and other information, resides in the first page-sized section of the file. The first text page image therefore starts at file offset 32768. As a result the normal minimum executable file size is 96 Kb, or more if a symbol table is present. However actual disc usage is not as high as this would imply because each page image requires a number of disc blocks (normally 4 or 8) to hold its data, and when the last page of each program section (text or data) is less than full, it consumes fewer real disc blocks – the remainder are empty 'holes'. The header only needs one block, so the remaining 3 or 7 blocks of the first 32 Kb file section are also empty.

3. Shared libraries

As part of the solution to the problems of disc space we developed a simple shared library mechanism. This saves both disc space, because each binary image no longer contains any library code, and physical memory space during program execution, because multiple programs share a

[1] Not to be confused with RISC/os, a product of MIPS Computer Systems, Inc.

single in-core copy of the library.

To summarise the scheme, any binary executable image may be linked with at most one shared library. This may seem a painful restriction but it is not quite as bad as it seems since that shared library may itself be built upon, and include the definitions of, another (and so on, up to a kernel-imposed limit, currently 8). The executable file header includes the textual name of the library image file (normally resident in */usr/lib*) and its binary timestamp. The former item is used to find the file, and the latter serves as a consistency check since it must match another instance of the timestamp located in the library image file's header. If the library image file is not found or the timestamps do not match, the program cannot be run[2]. If everything is acceptable (including the location and timestamps of any further nested libraries), the library image file is then associated with a text structure; shared library text sections are demand paged just as for ordinary programs. To save precious physical memory, shared library data is not allocated its own data page(s). Instead it resides at the very start of the normal user stack region (displacing the real user stack base as required), and is loaded immediately at *exec* time. As with program images, all processes using a particular library image share a single in-core copy.

The shared library mechanism currently lacks refinement in that library images cannot usually be replaced with new versions, e.g. to fix bugs, in such a way that already-linked programs will benefit. The only possible way to do it is by binary patching of an existing library image. Programs must normally be relinked with the new library instead. For the sake of old programs, an earlier library image file may be retained for as long as necessary, since the normal shared library naming scheme is based upon the date and time it was created, ensuring uniqueness.

Although the requirement to relink for library updates seems rather painful, this is really no worse than having no shared libraries at all, and the benefits in other ways are substantial. Especially for large libraries, such as those used with the X Window System, the scheme produces a very significant disc space saving (see §9.2). It also permits much better performance, by reducing the contention for limited physical memory, when multiple programs using the same library are running at once.

4. Code compression

During development of the RISC OS operating system, some work was done to tackle the problem of program size on disc. The obvious approach is to compress the data. However, although much research has been done on both generalised and specialised data compression in other areas, little reference could be found to compressing code intended for execution, so the path was rather unclear. Nonetheless a code compression and decompression technique was developed which is tuned to the general characteristics of the ARM instruction set; without excessive effort, file size reductions of around 50% for compiled programs were achieved in the context of RISC OS. We normally refer to the encoding process as 'squeezing' rather than 'compression', simply to differentiate it from conventional use of the latter term which has a more general scope. In addition, 'compress' is already generally known as the name for a common implementation of one standard algorithm.

In RISC OS, a whole program is loaded at once, into program-writable memory, and remains resident until completion. There is no special alignment of items within the image file: all the code and data is packed together contiguously. Squeezing involves analysing the pattern of 32-bit values found in the image file, producing two tables of common values, then encoding the program file accordingly, appending a compact form of the tables and a small piece of unsqueezing code. The program entry point is adjusted to refer to the extra code, which runs as soon as the program is started, copying itself and the data tables onto the stack out of the way, unsqueezing the main image and passing control to the original entry point.

---

[2] For this reason, critical booting and system-maintenance programs, which may be needed when recovering from file system problems, are normally linked with the unshared library.

5. Code squeezing in RISC *i*X

We are not aware of any previous application of the concept of data compression to binary executable files under UNIX. It seems that others aren't either; to quote from a regular contributor to the USENET newsgroup comp.unix.internals:

> I once suggested to Chris Torek that the kernel should execute compressed programs. He groaned.[Cot91]

Chris Torek subsequently provided an explanation of why it is not feasible to use conventional data compression algorithms in this context[Tor91].

For RISC *i*X we adapted the original scheme somewhat. We wanted to preserve the ability to load individual pages on demand rather a whole program at once. We also wished to avoid self-modifying code and the need to make the text area writable, which would preclude sharing texts between processes. In the adapted scheme, we perform analysis globally on the combined text and data areas, requiring only one pair of compression data tables for each program, but squeezing individual page images separately. Each squeezed page image starts with a very short header which defines the size of squeezed data it contains and the length of the data when unsqueezed, along with a checksum. The kernel itself performs the unsqueezing, in the code which handles demand page loading.

In the unsqueezed format each full page as stored on disc would normally occupy 32 Kb, or 8 blocks on a file system with a 4 Kb blocksize[3]. Squeezing reduces the data to around 16 Kb on average, taking up only 4 or 5 blocks. All the space at the end of each squeezed page is logically unused and hence can be stored as 0. This is where the design of the UNIX file system comes to our aid (as it has already done to a lesser extent, for partial unsqueezed pages). Any block which is all zero can be represented in the file system by a disc block number of 0; it then occupies no disc space. The *lseek* system call is used by the squeeze program to create such empty blocks. It simply moves the file pointer past the unused space before writing the next non-zero datum.

The squeezed executable format therefore comprises an extended header, then the text pages, data pages, and an optional symbol table. The header includes a new magic number (QMAGIC) to identify the squeezed format, and is extended with the two compacted tables of values for the unsqueezing algorithm. As before, it is zero padded to the next page boundary. Each page of text or data is squeezed in place, i.e. the squeezed data starts at the same offset in the file as the original unsqueezed data, and the symbol table if present also starts at a page-aligned offset. The holes in a squeezed file are thus interspersed, occurring at the end of each page of data.

It might be thought more space efficient to pack all the pageable data down into a single chunk, but this would rather complicate the job of loading an individual page. In addition it would mean that some programs which process executables would have to be modified to understand multiple formats, whereas apart from the header extension and the actual contents of each prototype page, squeezed and unsqueezed programs appear the same on disc.

6. The squeeze program

Squeezing is applied to a normal demand-paged format (ZMAGIC) program as a separate operation. The linker produces normal unsqueezed output, but may immediately invoke the squeeze program if requested via a command line switch. To more readily understand the following outline of the squeezing algorithm, it is probably helpful to look at the code for the corresponding unsqueezing operation, given in the appendix.

---

[3] When squeezing was first introduced, the blocksize was constrained to be 8 Kb; moving to 4 Kb blocks noticeably improved the disc space saving because of the finer allocation granularity.

6.1. Outline of operation

In operation, *squeeze* loads the complete program image into memory and analyses the contents of each page. Firstly, zero words at the end of a page are discounted: the kernel applies the rule that all data in a page beyond the unsqueezed section is initialised to zero. Next a list is constructed, from the remaining data words in the whole file, of all distinct 32-bit values except 0. Each list item includes the number of occurrences of the value.

The list is then sorted in frequency order, and the most commonly occurring 1792 values are inserted into a table in ascending value order and their entries removed from the list. Each occurrence of one of these values in the original data will be encoded by its index in the table. One of the seven values 9..15 in a 4-bit control item in the squeezed data stream is used, in conjunction with a single following byte in the stream (1792 = 7 × 256). Each 32-bit value so encoded is thereby squeezed from 32 bits down to 12.

If any values remain on the list[4], a new frequency list is constructed. All remaining items in the original list are shifted right by 8 bits, removing the lowest byte of each value and placing 0 bits into the top of the word. The frequency of each resulting value in the top 24 bits of the remaining data words is then computed. Again, the first 1792 entries in the sorted list are extracted and then sorted by increasing value into a second table. To encode any word not occurring in the first table but whose top 24 bits are found in the second, one of another set of 7 values (2..8) in the 4-bit control field is combined with the first of two following bytes to create an appropriate index into the second table. The lowest 8 bits of the original 32-bit value are supplied in the other byte. Thus these words reduce in size from 32 to 20 bits.

Occurrences of the value 0 within a page's data are encoded in the squeezed format by a 4-bit control code of value 0. This represents an 8 to 1 reduction by encoding (but see below).

Any other word value encountered in the program image is encoded in 36 bits: an initial 4-bit control code with value 1, followed by four bytes in the squeezed data stream. These must be assembled to recreate the word during unsqueezing.

Once all the tables have been constructed the actual encoding of each word of the page image is performed. Each original data word is re-read and checked (using hashing for speed) to determine which encoding is appropriate. If necessary, the required index value is also determined. A suitable sequence of items is then added to the squeezed data stream for the page in which the word lies.

In fact, encoding is actually done on a pair of words at a time: the first byte of the resulting squeezed data stream contains *two* control values, and any additional byte codes (from 0 to 4 for each value) follow that, in order.

Finally, the tables themselves are compacted into another, byte-oriented, stream format in which items encode the *differences* between successive table entries. Since the tables have already been sorted, deliberately for this reason, the differences are as small as they can be, and hence the encoding is very efficient. The largest compacted pair of tables amongst all the executables in the current release of RISC *i*X requires 6115 bytes in total. However even that represents a compression ratio of over 51%, after taking into account that entries in the second table are effectively only 3 bytes long.

6.2. Rationale

Many 32-bit values do occur repeatedly in program files. Examples of this include the particular fixed (or only slightly varying) instructions planted by the compiler for such things as procedure entry sequences, and address constants for frequently referenced data objects (e.g. the value of *stderr*, or *errno*). Therefore, encoding very common values such as these in a table is likely to save considerable space.

[4] There may not be any: small program images often contain less than 1792 distinct word values.

The reason for basing the second stage of analysis on the top 24 bits of the remaining words arises from the fact that most program image words are actually ARM instructions. These have a format such that the lowest 8 bits are often part of a constant field of some sort (e.g. a branch or memory addressing offset) and tend to vary more widely than the higher order bits. Thus there is a greater chance of finding multiple word values with the top 24 bits in common than for other 24-bit subsets of a word. Notably, the top 4 bits of every ARM instruction are a condition code controlling the circumstances in which the instruction will be executed. In most compiled instructions, this field has the value 0xE meaning 'always'; hence the variation in these particular bits is especially low.

Other, non-instruction, words in a program image are frequently address constants, and again the variation in the lower bits for these is much greater than in the higher ones, since most objects thereby addressed reside within either the data or bss program sections (or in the shared library data area). The range of addresses these sections extend over is usually encompassed by varying a relatively small number of lower order address bits.

The justification for specially encoding 0 as a single 4-bit control value may at first sight seem obvious. In the original context of squeezing, this is indeed very important because RISC OS has nothing corresponding to UNIX's *bss* (a section of memory at a fixed address, initialised to zero before program startup, which takes no space in the image file). All statically allocated data must normally be represented in the program image, and hence there is often quite a large quantity of zero data: the 8-to-1 reduction helps greatly[5]. However under RISC *i*X it turns out that once the zeroes at the ends of pages are eliminated, there are in fact remarkably few occurrences of zero in most program images. Thus in adapting the concept to UNIX there might well have been an advantage in changing the encoding, for example to extend one of the tables by another 256 entries and remove any special treatment of 0.

6.3. Unsqueezing

A much simpler *unsqueeze* program performs the reverse operation to *squeeze* when required e.g. for debugging purposes. To ensure consistency the same source code for the central unsqueeze operation is used when compiling both the kernel and the unsqueeze program.

7. Kernel processing

The primary changes made to our port of 4.3 BSD in order to support execution of squeezed images occur in the kernel. In particular, the modules which implement the *exec* system call and the handling of page faults during execution of demand-paged programs required most modification.

7.1. General operation

The kernel recognises squeezed executables by the magic number at *exec* time, and reads in the two unsqueezing tables in the header at this point. They are immediately expanded out from their compacted form (§6.1). The expanded tables occupy on average 5-6 Kb of memory, limiting at 14 Kb for larger programs. To save memory, since most programs use much less than 32 Kb of stack, the expanded tables are kept in user space just before the start of the user stack. Most executing programs never require more than just this first page of the stack region.

Also at *exec* time (or more specifically on the first *exec* of a program not already in the text cache), a table is constructed mapping each page to the disc block numbers for its disc-resident prototype. The kernel's internal *bmap* function (or in reality the relevant file system's specific implementation of the *bmap* vnode operation) is repeatedly called to obtain this information. How many disc addresses are required per page varies according to the block size of the particular filing system (4 Kb or 8 Kb). This table is then associated with the text structure, to save time on

[5] This probably partly explains why squeezing seems to give greater space savings in its original context than under RISC *i*X.

subsequent execution, and will be looked up on each page fault requiring a disc access.

To load a page on demand, the kernel reads in the blocks containing the squeezed code or data directly into the new page frame (not through the buffer cache in general, but see §7.4) then applies the unsqueezing algorithm on the data. Since the data starts at the same file address regardless of squeezing, there is no problem finding it. The page read operation takes advantage of any adjacency of disc blocks[6] and of course holes take no time to load! A checksum kept with the squeezed data helps protect against problems such as the possibilities - practically never seen - of a wayward program corrupting the unsqueezing tables in its own stack area, or of the executable image file itself being corrupted.

7.2. Unsqueeze algorithm

A slightly simplified version of the kernel's unsqueezing routine is given in the appendix. (When reading the code, note that a C **int** occupies 32 bits on ARM.) The unsqueeze algorithm and squeezed data format are themselves carefully tuned to the capabilities of the ARM architecture. The processor can directly load and store 8 and 32-bit quantities, but not 16-bit ones[7]. In addition the addressing modes allow the base register to be modified as part of a load or store operation, providing a more general form of the auto-increment/-decrement modes of other processors.

These features are clearly reflected in the data format and in the unsqueeze code: the input is an array of bytes, the output is an array of 32-bit words, and both are accessed sequentially using C's **--p* expression form. All accesses to the unsqueezing tables are as words, and no **short** (16-bit) objects are used at all.

Note that unsqueezing is performed backwards, since the squeezed data is loaded into the base of target page and overwritten during the process. Two pointers are used, tracking the next input and output items. When they come near to colliding (since by the nature of the process the output pointer must move faster than the input pointer) the remaining input data is copied into a separate small buffer and read from there. In pathological cases (i.e. programs with peculiar patterns of data) this buffer may not be big enough. In such a situation the unsqueeze operation fails and the process will be killed with an error message. A modest modification of the algorithm would permit this situation to be recovered from, but since the problem has never been observed in the production system, there seems no need.

7.3. NFS and demand paging

Under NFS, file blocks as seen by a client machine are simply sequential numbers relative to a vnode which represents the file. Block 0 is the first actual block of a file, and there is no explicit representation of holes. The code to load in a page from a prototype file will therefore not be able to optimise the reading operation in the way that the local file system permits. Discless machines page programs in via NFS over Ethernet, which is typically somewhat slower than standard discs, so we wanted to handle this case as well. We have therefore added a heuristic to the page loading code which notices that all the blocks for the page are present and numbered sequentially (suggesting that the file is stored on NFS or a similar type of file system); it therefore reads just the first block, containing the page header, to find how much squeezed data there really is, then fetches only as much as is required. The table of block addresses for the page (associated with the program text structure) is updated appropriately so that subsequent faults on the page can be processed without extra cost.

The result is that loading a squeezed program under NFS is often quicker than loading the unsqueezed version, since there is a much greater overhead per byte of data traffic through the

[6] With suitable tuning of the file system disc layout parameters, this happens most of the time in the standard distribution.

[7] Half-word loads are possible, using a normal word load on any 2-byte boundary, but the upper 16 bits of the loaded register end up containing junk which normally must then be cleared out. Half-word stores must be implemented using two byte stores.

network layers than with disc transfers. For example, starting up a squeezed copy of GNU EMACS (640 Kb or 20 pages) for the first time, over NFS, was timed at an average of 5.0 seconds when unsqueezed and 3.9 seconds when squeezed. The reported system time during this operation however went up from 0.8 seconds to 1.1, reflecting the cost of the unsqueezing operation involved. In this case, most of the pages are full of code or data and the average squeezed page size is about 17 Kb. The gain is even more marked when a page is mostly empty, although this typically occurs more with small programs which are quicker to page-in anyway. As an extreme case, a program was contrived which measured the real time required to touch (and thereby force the page-in of) 10 data pages, each containing only a few bytes of non-zero data. The time to do this diminished from 1.08 seconds to 0.30 seconds after squeezing. The system time also went down, from 0.45 to 0.1 seconds, since the unsqueezing time is negligible and per-byte network processing time dominates. Of course there is another – hidden – gain involved here in that network traffic is reduced, increasing the bandwidth available for other uses.

7.4. Data page-in

As in standard 4.3 BSD, the kernel maintains a hash table for text pages which have become free, so that frequent execution of a program can reclaim these without disc access. However data pages cannot be reclaimed since in general they will have been modified. In addition the page-in operation is conventionally performed directly into the target physical page rather than via the buffer cache. This means that a disc access will always be required for a data page even if the program was recently run. For slow discs or other slow media we would really like to avoid a disc access if possible. Most program data areas are less than 32 Kb; indeed, many are only a few Kb. Therefore if all the data will fit in a single 8 Kb buffer, it is loaded via the buffer cache. This has an impact on other users of the buffer cache but the benefits were thought great enough for it to be worthwhile in general use; the option can however be disabled if required.

7.5. Swap space

As a means of further reducing disc requirements, but in contrast with standard 4.3 BSD, program texts under RISC *i*X are not cached on the swap area (other than when debugging). Instead we simply reload text pages from the prototype file whenever a text page fault occurs. The saving in required swap space is much more important than the overhead of going through the file system for page reloads, and of course with such large pages page faults occur less frequently.

8. Other system changes

Because of the 32 Kb page alignment most binary executable files, even unsqueezed ones, are typically quite sparse, so problems can arise when moving them. The utilities *cp*, *mv* and *tar* (amongst others) have therefore been modified to create holes in output files wherever possible, by seeking rather than writing when the buffer to be written is all zero. (For compatibility the original copying behaviour is still available if required.) This preserves the disc usage of executables when they are copied around; the block size of the target file system is checked to decide what would constitute a hole in the destination file, so the created file always occupies the minimum amount of space.

A modified version of *fsck* can be requested (via a new *–Z* flag) to perform an extra pass, to turn blocks containing all zeroes into holes. This takes a long time, but is sometimes very worthwhile. We have recent experience with an operating system test suite which uses its own file copy program to move the compiled test programs into place. Even though the squeeze option was used during compilation, **all** holes were filled in by the copy. Subsequently an extra 35 Mb (out of around 80 Mb) was recovered by means of *fsck –Z*.

Debuggers cannot directly read a squeezed image file, although the symbol table is still available: normally the *unsqueeze* utility is run before debugging. As an alternative it would have been possible to add the unsqueezing routine to each debugger (*adb* and *dbx* are provided with the system) to permit examination of prototype code and data. Since it would still not be possible to make arbitrary modifications to a squeezed file (resqueezing would be required in general), the increase

in debugger complexity was not considered worthwhile.

In previous releases of RISC *iX* we have also applied conventional data compression techniques to the on-line manual pages, again in order to reduce disc space requirements. This does impose a slight speed penalty, and for the latest machines with a somewhat larger disc the space saving was not considered significant enough to warrant this.

As a final touch, the kernel image file */vmunix* is itself squeezed; the bootstrap code includes the unsqueezing routine, applying it to each page in turn as normal. In consequence, although the latest production kernel has a nominal size of 784 Kb, it occupies only 540 Kb on disc, a useful saving. The percentage space saving (31.1%) is smaller than it might have been because the full kernel symbol table of around 60 Kb is retained with the file for use by system diagnostic programs such as *ps*.

9. Performance

The squeeze/unsqueeze algorithm is known to give good results in its original context of RISC OS: 45-50% disc space saving is normal, and program loading is overall quicker by as much as 40 percent, depending on the storage device data rate. There are a number of factors which tend to reduce the benefit in the UNIX environment. Nonetheless the results were found to be worthwhile.

9.1. Disc usage: theory

The space reduction obtainable from the squeeze algorithm is limited at the extremes. For small programs this arises from the relatively coarse granularity of disc space allocation in the Berkeley Fast File System[McK84]. Where individual page images have little data in them, there is often no saving at all. In particular if every page image fits in a single disc block, no reduction is possible; in pathological cases the file could actually grow, although such a case has never been observed. Obviously in this case squeezing is not useful in respect of space savings, however we have already shown (§7.3) that other benefits may be obtained nonetheless.

At the other end of the scale, the disc occupancy reduction possible for very large programs is limited because the tables of unsqueezing data have a fixed maximum size. In large programs there may be so different frequently-occurring values that they cannot all be represented in the tables. For the residue, the space taken in the squeezed file for each 32-bit output word is 36 bits; this reduces the effect of the savings achieved for the data which was represented in the tables.

9.2. Effect of shared libraries

Earlier (§3) we described how the sharing of library code saves disc space and physical memory. In practice, since the use of shared libraries makes individual program files smaller, the gains from code squeezing diminish if the two techniques are used together. In order to assess what gains would have been possible from squeezing alone, a subset of 453 programs out of the 563 executables supplied in RISC *iX* 1.21 were built using non-shared libraries and then squeezed. Figure 1 shows the results: note that the size of the main shared C library (216 Kb) is not included in the figures.

	Unsqueezed (Kb)	Squeezed (Kb)	Reduction (%)
Unshared	30292	19796	34.5
Shared	11168	8928	20.1

Figure 1. Effect of shared libraries

As can be seen the use of shared libraries clearly has the larger impact on disc space: a net saving of 62.4% is achieved for the sample set, when the shared library image file is included. For various reasons it was not possible to include any of the X11 programs in the sample. Since these are larger than average, the overall improvement from squeezing, for both the shared and unshared library cases, is actually somewhat better than appears from the table.

9.3. Disc usage: final system

Figure 2 shows the disc occupancy for the kernel, a few individual files, the main utilities directory */usr/bin*, the X11 binaries directory */usr/bin/X11*, and the full set of 563 distinct executables provided in release 1.21 of RISC *i*X, with and without squeezing. (The kernel is not included in the full set.) The relative modesty of the savings in */usr/bin* and the overall 27.7% figure reflect the fact that there are many small programs in the distribution, like *od* and *ln*, whose disc size does not change when squeezed. The better result for the X11 programs is a reflection on the size of the average X program, despite the use of shared libraries!

Item	Files	Unsqueezed (Kb)	Squeezed (Kb)	Reduction (%)
/vmunix		784	540	31.1
/usr/bin/awk		64	40	40.0
/usr/bin/cat		16	16	0.0
/usr/bin/csh		104	64	38.4
/usr/bin/ex		192	112	41.7
/usr/bin/find		28	20	28.6
/usr/bin/lex		48	28	41.7
/usr/bin/ln		16	16	0.0
/usr/bin/od		20	16	20.0
/usr/bin/troff		80	44	45.0
/usr/new/emacs		728	456	37.4
/usr/bin/*	132	3484	2744	21.2
/usr/bin/X11/*	97	5160	3336	35.3
/usr/ucb/*	65	1624	1300	20.0
Overall	563	19940	14416	27.7
Average		35.4	25.6	27.7

Figure 2. Disc usage: final system

9.4. Unsqueezing time

Instrumentation was added to a kernel to determine how much cpu time is spent in the unsqueezing routine. Measurements given here were obtained using an Acorn R260 workstation which uses the second generation of ARM processor and is considerably faster than the original R140.

Final size (Kb)	Unsqueeze time (ms)
0 .. 4	2.1
4 .. 8	5.1
8 .. 12	7.9
12 .. 16	9.4
16 .. 20	11.2
20 .. 24	12.6
24 .. 28	17.3
28 .. 32	20.3
32	21.4

Figure 3. Unsqueezing time

The average time to unsqueeze a page is shown in Figure 3 as a function of the final size of the page. As described above, all zeroes at the end of a page are ignored when the squeeze operation is performed, and each page contains a header specifying the end of non-zero data. Many page faults require less than 32 Kb of data to be unsqueezed.

From this data the unsqueezing rate is found to be about 650 nanoseconds per output byte, 2.6 microseconds per output word, or roughly 1.5 output Megabytes/second. This is obtained from a version of the unsqueezing routine a little more tightly (and un-prettily) tuned than the 'reference' version given in the appendix. The performance difference is around 20%, although it is believed that further slight speed-ups are possible[8].

9.5. Total page fill time

It is easy to measure the time taken to satisfy an individual page fault. What is not so clear is what the result means. The overall time to fill a page from the file system includes the I/O latency and the unsqueezing time. However the I/O latency is affected by a large number of factors, including basic device access times and device buffering mechanisms. Most important is how busy the device and the processor are at a given time, in terms of the number of requests and runnable processes waiting in their respective queues. In addition, both the I/O latency and the unsqueezing time vary according to the amount of data in the page, so the actual characteristics of the set of programs in use will affect the result.

Despite these problems, some experiments have been done to measure the minimum I/O latency for transfers of various sizes. This permits putting a reasonable lower bound on page fill time, from which comparisons can then be made with the unsqueezed case. The tests were carried out on an Acorn R260, fitted with a 100 Mb SCSI disc. Obviously the results are therefore specific to this particular hardware combination. Testing was somewhat *ad hoc*: it involved simply running as many different programs as possible as often as possible, in order to generate as many page faults as possible(!). Since we are looking for a minimum and not some sort of average time, the seeming randomness and unrepeatability of this approach is not a concern. To maximise the number of page faults causing reference to the disc, buffered data page reads (§7.4) and the reclaim of text pages from the free page list were both disabled.

In summarising the results we consider just full 32 Kb pages. The minimum I/O latency observed for a single contiguous transfer of 16 Kb (out of 557 occurring during the testing period) was 30.5 ms. The minimum latency on a single transfer of 32 Kb was 64 ms. Now the average squeezed size of a full 32 Kb page is 15.5 Kb (measured from the set of 563 executables in RISC *i*X 1.21), and of the 179 such pages in the set, only 39 squeeze to more than 16 Kb, so most squeezed full pages require only a 16 Kb transfer, taking a minimum of 30.5 ms. In their unsqueezed form such pages require either a 32 Kb transfer or two or more smaller ones (since they are less likely to be contiguous on the disc), taking at least 64 ms. Therefore for full pages it is usually *quicker* to fill from the squeezed form of the page than from the unsqueezed form. When the I/O latency is higher than this minimum, the relative real time cost for the unsqueezed case is even greater[9].

This is not a completely free speed-up of course: the cpu is dedicated to the unsqueezing operation for part of the time, rather than being available to run other processes while the data is in transit. However there are hidden factors which compensate in part: more work needs to be done in the unsqueezed case, both in setting up the disc subsystem for the extra data transferred, and in performing the data movement from device hardware buffers to memory[10]. Exact costs for the latter activity are hard to measure and vary somewhat depending on the specific device hardware, but a rough approximation of the average cost yields a figure of around 0.4 cpu-milliseconds per Kilobyte. Thus although unsqueezing a full-sized page from 16 Kb of squeezed data takes around 21 ms, 6.4 ms of cpu time are saved by not transferring the other 16 Kb from the disc, apart from a

---

[8] The unsqueezing routine used by squeezed programs under RISC OS runs considerably quicker because fewer consistency checks need to be applied. In RISC *i*X the kernel's memory could be overwritten if the squeezed data were inconsistent, whereas under RISC OS a program unsqueezes itself, in user mode.

[9] For what it's worth, the respective average latencies for 16 and 32 Kb transfers during the tests were around 65 ms and 124 ms.

[10] This is because the systems on which RISC *i*X runs do not have DMA: all bulk storage device data movement is performed under block- or sector-level interrupt by the processor.

possible reduction in transfer set-up costs.

9.6. Performance Summary

We have found that disc usage is significantly improved, and page fault delays may actually diminish. But the reduced volume of data being transferred from the file system per page fault provides a further advantage, as has already been hinted at – the overall level of activity on storage devices caused by paging is also reduced. Hence the file system bandwidth available for other purposes, principally normal file access, increases. This can be particularly helpful in networked use (§7.3).

As a counterbalancing point, it should be noted that with such a large page size, and because of such things as text page reclaim from the free page list, paging from the file system is a relatively infrequent event anyway (unless the system is thrashing). Thus the actual degree of improvement in available file system bandwidth is not particularly large. On the other hand, the overall additional cpu-time cost of executing squeezed as opposed to unsqueezed programs is therefore also relatively insignificant.

10. Current weaknesses

The system in its current state works well, and has clearly achieved its original goal of reducing disc space usage. However, some choices were made during development which have proved less than optimal. Most of these mistakes occurred because the original squeezing technique was not critically reviewed in relation to its use under UNIX before the initial porting work was done. In addition, insufficient thought was given to alternative methods of reducing program size, such as changing the alignment of page images (§11.1).

The first significant slip was the decision to retain compatibility with the original unsqueezed file format: this turns out to cost disc space. If each page is aligned on a 32 Kb boundary within the file then the normal minimum file size is 96 Kb. With a 4 Kb file system block size, all executables therefore require an indirect block; also, the last block is always full sized rather than potentially being a fragment. Although changing this would complicate slightly the coding of the kernel and of utilities which read an executable file's symbol table (e.g. *nm* and *ps*) in retrospect it would have been very worthwhile.

Secondly, too little account was taken of the file system disc allocation details. Many pages have only a few Kb of squeezed data, and for these the 4 Kb allocation granularity (in conjunction with the large page alignment) significantly impacts on the achievable space saving, particularly since most files have only a small number of pages.

Other considerations, such as the layout of squeezed data within each page image, and the possible differences in optimal squeezed data encoding, were also given less than maximum thought. The next section considers improvements which might be made in all these areas.

11. Possible improvements

It is interesting and useful to investigate various adjustments to the squeezing/unsqueezing scheme. What the analysis has revealed is that the main factor limiting possible space savings is the format of the squeezed data file. A further study has therefore been performed to assess how changes to this format might help. Calculations were performed on the same set of 453 executables as was used for the earlier consideration of space savings.

11.1. Page alignment

The primary factor is the alignment of the page data within the image file. As noted above, 32 Kb page alignment has a serious effect upon the disc space consumed. A substantial gain is obtained by using a smaller page alignment. In the existing scheme all page images are the same length in the file, whereas with a smaller alignment squeezed pages would tend to vary in length. Thus a table of page start offsets would need to be added to the file header, to allow the kernel to find the relevant data quickly on a page fault. For alignments less than the file system block size,

further small complications in the page-in code become necessary to cope with partial blocks. Also, the performance might fall in this case because it will sometimes be necessary to read more than the minimum number of blocks for the size of the page data.

Simple experiments were done to ascertain the exact effect of page alignment reductions. Three plausible possibilities were tested: 4 Kb, 2 Kb and 1 Kb alignment. These values were chosen because they are whole multiples of common physical disc sector sizes, ensuring that there is no difficulty in reading the required data directly from the disc into place in the target page; a size of 512 bytes was not tested because at least one device which our system supports uses a 1 Kb sector size. In addition, the extreme of using 4 *byte* alignment was also tested. This is considered to be impractical for normal use – the page-in code would have a lot of extra work to do, including copying the squeezed page data to the page start before it could be unsqueezed – but it is obviously a possible approach if absolute minimum disc space usage is required. Figure 4 shows the resulting size of the full set of RISC *i*X 1.21 executables, as released (32 Kb alignment) and with these variations. All measurements are based on the full set of RISC *i*X 1.21 executables (excluding the kernel) and assume a file system using 4 Kb block size, 1 Kb fragment size.

Page alignment	Unsqueezed size (Kb)	Squeezed size (Kb)	Reduction (%)
32 Kb	19940	14416	27.7
4 Kb	17980	12332	31.4
2 Kb	15394	9542	38.0
1 Kb	14144	8411	40.5
4 bytes	13345	7676	42.5

Figure 4. Effect of page alignment

As may be observed, if we had used a 1 Kb page alignment for the file format in the first place, we would have directly achieved the same gain now obtained using squeezing. What is also worthy of note is that as anticipated, the effectiveness of squeezing increases as the page alignment is reduced. However, as file sizes diminish, the file system allocation granularity becomes significant. This explains the relatively small improvement when moving from 1 Kb page alignment to 4 byte page alignment. Clearly, there is little point in adding further substantial complexity to the system for such a modest gain.

As a final, and striking observation, consider this. The sample set of 453 executables used in other measurements, when built using the original format, unshared library and without squeezing, occupies 30292 Kb. If all the files were rebuilt with the shared library, and squeezed using the proposed 1 Kb page alignment, the space requirement would drop to a mere 4013 Kb, a factor of better than 7.5 to 1!

11.2. Squeezed data format

It would be possible to speed up the unsqueezing routine a little if the layout of the squeezed data for an individual page were altered. The original squeezed data format was geared to unsqueezing a (potentially large) whole program at once, using the minimum amount of additional memory for workspace. When unsqueezing only a page at a time, one extra page could be used so that the unsqueezing process writes into a different page from that containing the squeezed data. This would avoid the problem of overwriting, and allow a different ordering for the items in the squeezed page image, to permit quicker unsqueezing. The squeezed data could be grouped into four separate blocks in each page image according to size: 4-bit codes, 1 byte codes, 2-byte codes and 4-byte codes, each block being aligned on, and padded to, a suitable boundary. A single memory access would then suffice for each type of item, rather than having to construct the larger

items using individual byte loads[11].

Another beneficial change to the squeezed page format would be to remove each page header (which occupies only 8 bytes) and locate it in the main program header instead. Doing so would remove the need for a table of page-start offsets if the page alignment were reduced, since the same information can be computed from the individual squeezed page sizes. Alternatively, it would mean that for the case of paging over NFS (§7.3), the kernel would not need to specially fetch the first block of a page to discover how much squeezed data there actually is.

11.3. Shared squeeze tables

One change to the original squeezing algorithm which has been proposed involves performing global word frequency analysis across many executable programs at once, rather than one at a time. Assuming that there is a high correlation between the most frequently occurring values in individual files, the table of values so produced could be shared between them. This would save space, especially relative to the current format where of the 563 executables in the full set, 139 have compacted squeeze tables which overflow into a second 4 Kb block in the extended header. If all tables fitted inside the first block of the file, we would gain over half a Megabyte.

There are obviously a range of different ways in which such a scheme could be implemented. One grand analysis might be performed across all available programs; a certain number of the most common values would then be fixed for all time into a table. This would be kept in the kernel, and its contents made known to all the relevant utilities. Such a scheme would only work if the set of the most common values were highly correlated between programs; it is also assumed that it would not change over time. The latter assumption is unsafe given likely developments in compiler technology, since 'typical' compiled code instruction patterns may vary.

Alternatively, (large) groups of programs sharing similar squeeze tables could have the most frequent common entries extracted and kept in a file in a standard directory, in a scheme analogous to shared libraries. The kernel would pick up (and cache) the required table using a filename string in each executable program's header. This scheme would allow tracking of changes in the pattern of data value frequencies.

11.4. Per-page squeeze tables

Moving in the opposite direction from shared squeeze tables, it may perhaps be advantageous to build a small table of particular values for individual pages within one program, as well as the existing per-file table. This might allow files with unusual word-frequency characteristics (usually large files with significant amounts of data, such as EMACS) to be squeezed more effectively.

12. Applicability

The key features of the system which led to the design described here are:

- disc space is a very limited commodity
- the page size is greater than the file system block size
- machine instructions are all one word long

The technique might be applied beneficially to other systems with similar characteristics. Small personal machines with modest disc space, like Acorn's own systems, are obvious targets.

If the file system block size is smaller than the minimum of 4 Kb imposed by the Berkeley Fast File System, then the minimum hardware or logical page size at which the basic technique becomes viable also goes down. Even with smaller pages, it might be feasible to arrange that all page-ins from the prototype file occur in larger units, so as still to allow disc space savings using a similar method.

[11] This is necessary in the current algorithm, because the 2- and 4-byte squeezed items are arbitrarily aligned and ARM, in common with most other RISC processors, does not support the reading of misaligned data.

Alternatively, dispensing with page-alignment in the program file as discussed above would permit the use of squeezing in a more general context. however if the paging unit is around the same size as the normal physical sector size, there is less likelihood of benefit.

The regularity of the format of page images (viz. all code and most data items are 32-bit aligned) is very important in allowing a simple compression algorithm with very fast decompression. It is not clear that a scheme could be as easily devised to allow the rapid decompression of less simply-structured data such as the variable-length instructions of most CISC processors.

13. Conclusions

In the context of the original requirements, through the use of shared libraries and squeezing we have been able to produce a small and cheap system which nonetheless contains a substantial volume of software. However the benefits have turned out to be rather broader in scope.

The current method of squeezing individual pages actively exploits what would otherwise be a serious disadvantage of the hardware architecture: the page size is larger than the file system disc block size. However it has been found that this relationship is not essential to the scheme; alternative image file arrangements which did not assume it would actually permit greater space savings to be obtained, albeit at a possible cost in speed.

In summary, the technique described in this paper:

- saves considerable amounts of disc space; this is most useful for systems with small discs.
- reduces the amount of data transferred from the file system, which improves disc availability and/or reduces network and fileserver load.
- saves wall clock time when paging from slower storage devices or heavily loaded fast ones; sometimes the reduction is humanly perceptible.

Although in its current implementation squeezing alone has turned out to save somewhat less than 50% of disc space, the overall savings we have achieved perhaps justify the claim that you can, after all, fit a quart into a pint pot.

14. Acknowledgements

Many people have been involved in the work described here: I am indebted to them all. Richard Cownie designed and implemented the original squeeze and unsqueeze mechanism, and Keith Rautenbach and Andy Bray performed the initial adaptation for RISC *i*X. John Bowler designed and developed the shared library mechanism. Lawrence Albinson was largely responsible for the original RISC *i*X port; he and David Lamkin provided a strong lead for the collective effort. Mike Challis has been very supportive of both the production of this paper and the whole RISC *i*X project.

I am grateful to Gretchen Phillips for her comments on a late draft of the paper; she also very kindly printed the final copies. Finally, my thanks must go to Barry Shein: without his suggestion I would not have written this paper at all.

References

[Cot91] J. Cottrell (Root Boy Jim), *comp.unix.internals*, <118868@uunet.UU.NET>, January 1991

[For90] C. H. Forsyth, "More Taste: Less Greed? or, Sending UNIX to the Fat Farm", *Proceedings of UKUUG Technical Conference*, London, July 1990.

[McK84] M. K. McKusick, W. N. Joy, S. J. Leffler, R. S. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, vol. 2, August 1984, pp. 181-197.

[Swa90] R. Swartz, "The Case Against UNIX Standards", *Proceedings of UKUUG Technical Conference*, London, July 1990.

[Tor91] C. Torek, *comp.unix.internals*, <9950@dog.ee.lbl.gov>, 15 Feb 1991

Appendix: unsqueeze code

```
#include <sys/types.h>
#include <sys/param.h>

/*
 * Format of an unsqueezing table as the kernel keeps it in memory.
 */
struct unsqueeze
{
    int sq_items;                /* Number of valid entries in table */
    int sq_table[7*256];         /* The table data proper */
};

/*
 * Different ways of looking at a page to be unsqueezed in place.
 */
union page
{
    unsigned char bbuff[NBPG];  /* as bytes (squeezed data in) */
    unsigned int wbuff[NBPG/4]; /* as words (unsqueezed data out) */
    struct                      /* as header+data */
    {
        struct sqheader
        {
            u_short rpoffset;   /* end of input data: in bytes */
            u_short wpoffset;   /* end of output data: in words */
            unsigned int check; /* additive sum of output words */
        } info;
        unsigned char data[NBPG-sizeof(struct sqheader)];
    } encoded;
};

/*
 * The primary unsqueeze operation is defined as a macro and in-lined
 * in the main unsqueezing routine as often as necessary. It uses
 * local variables of that routine: beware!
 */
#define UNSQUEEZE(code)                                      \
{                                                            \
        unsigned int nx;                                     \
        switch (nx = (code))                                 \
        {                                                    \
          case 0:                                            \
            break;                                           \
          case 1:                                            \
            nx = *--rp;                                      \
            nx |= (*--rp) <<  8;                             \
            nx |= (*--rp) << 16;                             \
            nx |= (*--rp) << 24;                             \
            break;                                           \
          case 2: case 3: case 4:                            \
          case 5: case 6: case 7: case 8:                    \
            nx = ((nx - 2) << 8) | *--rp;                    \
            if (nx > sq3->sq_items)                          \
                return "unsqueeze: bad sq3 item";            \
            nx = (sq3->sq_table[nx] << 8) | *--rp;           \
            break;                                           \
          case 9: case 10: case 11:                          \
          case 12: case 13: case 14: case 15:                \
            nx = ((nx - 9) << 8) | *--rp;                    \
```

```
            if (nx > sq4->sq_items)                       \
                return "unsqueeze: bad sq4 item";         \
            nx = sq4->sq_table[nx];                       \
            break;                                        \
        }                                                 \
        *--wp = nx;                                       \
        chksum -= nx;                                     \
}

#define XBUFSZ 256      /* size of copy buffer for last stage */

/*
 * Unsqueeze routine: returns either NULL (success) or an error string
 * to print before the process is forcibly terminated.  pp is the
 * address of the page containing the squeezed data.  sq3 and sq4 are
 * pointers to the tables of 3- and 4-byte values.
 */
char *unsqueeze (pp, sq3, sq4)
  register union page *pp;
  register struct unsqueeze *sq3, *sq4;
{
    register unsigned char *rp;
    register unsigned int *wp, chksum;
    unsigned char buf2[XBUFSZ];   /* copy buffer */
    unsigned char *wend;
    int shrink;

    /*
     * The output data must be an even number of words, and
     * the starting point must not lie beyond the end of the page.
     */
    if ((pp->encoded.info.wpoffset & 1) != 0 ||
      pp->encoded.info.wpoffset > NBPG/4)
        return "unsqueeze: page header corrupt";

    /* Set up the read and write pointers */
    rp = pp->bbuff + pp->encoded.info.rpoffset;
    wp = pp->wbuff + pp->encoded.info.wpoffset;

    /* Exceptionally, squeezed data can be larger than output data */
    wend = (unsigned char *)wp;
    if (rp > wend)
        shrink = rp - wend;
    else
        shrink = 0;

    /* Fetch the original checksum */
    chksum = pp->encoded.info.check;

    /*
     * Loop while the pointers cannot overlap in the next iteration,
     * and the read pointer hasn't got back to the start of the
     * buffer. If the latter test fails, corruption of the squeezed
     * data must have occurred.
     */
    while ((unsigned char *)wp >= rp + (2 * 4 - 1) && rp > pp->bbuff)
    {
        unsigned int byte = *--rp;
        UNSQUEEZE(byte >> 4);
        UNSQUEEZE(byte & 0xF);
```

```
    }

    if (rp <= pp->encoded.data)
        return "unsqueeze: squeeze data corrupt";

    /*
     * Prepare to copy the remaining squeezed data into our extra
     * buffer. Note that it is possible (but unheard of) for there to
     * be too much data to fit. This code could be changed to cope
     * with such cases (preferably in a manner other than just
     * increasing XBUFSZ) but there seems little need.
     */
    rp = &buf2[rp - pp->encoded.data];
    if (rp > &buf2[XBUFSZ])
        /* oh dear, oh dear... */
        return "unsqueeze: rp underflow (kernel bug)";

    /* Round up the copy size to words for speed (bcopy is a bit fussy) */
    bcopy (pp->encoded.data, buf2, ((rp - buf2) + 3) & ~3);

    /*
     * Now keep unsqueezing until the write pointer reaches the
     * start of the page or the read pointer hits the start of the
     * buffer. These conditions *ought* to occur together, but
     * for safety we must keep testing both.
     */
    while (wp > pp->wbuff && rp > buf2)
    {
        unsigned int byte = *--rp;
        UNSQUEEZE(byte >> 4);
        UNSQUEEZE(byte & 0xF);
    }

    /* Final consistency checks */
    if (rp != buf2 || wp != pp->wbuff)
        return "unsqueeze: pointer mismatch";

    if (chksum != 0)
        return "unsqueeze: checksum failed";

    /*
     * If the encoded data was larger than the decoded data, zero
     * the bytes of the encoded data which were not overwritten.
     */
    if (shrink > 0)
        bzero ((caddr_t)wend, shrink);

    return (char *)NULL;
}
```

Biographical Information

Mark Taunton was born and brought up in Scotland. In 1982 he graduated from the University of Edinburgh with a B.Sc (Honours) in Computer Science. Since then he has worked for Acorn Computers Ltd in Cambridge, England, where he is a principal software engineer. He has focused on programming language and operating system implementations, and was involved from the start with Acorn's development of UNIX systems. His current research interests include compiler technology, digital audio processing, and the implementation and practical application of functional programming languages.

Since moving to Cambridge, Mark met and married Hillary. They have two children.

Availability information

Acorn supplies a range of ARM-based UNIX workstations running under the RISC *i*X Operating System. For further information, please contact:

Acorn Computers Limited
Fulbourn Road, Cherry Hinton
Cambridge CB1 4JN, England.

Telephone (+44) 223 245200
Fax (+44) 223 210685

Experiences with Audio Conferencing Using the X Window System, UNIX, and TCP/IP

Robert Terek

Joseph Pasquale

Computer Systems Laboratory
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093

pasquale@cs.ucsd.edu

Abstract

We present a voice-quality audio conferencing system built using standard system software, specifically the SunOS 4.0 version of UNIX, X11, and TCP/IP on an ethernet. Since this software does not provide real time guarantees, we describe a simple strategy for recovering from missed deadlines by discarding "late" audio data. Using this strategy, conferences proceed without delay entering into conversations. To simplify application programming, a key part of the conferencing system is an X11 server modified to accept audio requests. This not only provides network transparent access to audio devices, but also has implications for synchronizing audio with graphics or any other media that an enhanced X11 server may process. In this study, we describe the structure of the conferencing application and X11 server and then characterize the performance of this system.

1. Introduction

An increasing number of desktop workstations are being produced with some type of audio device. For instance, the Sun SPARCstation provides a telephone quality audio device, while NeXT workstations implement CD-quality audio using a DSP. These devices are being used to enhance the user interface with chiming clocks and other simple effects, but they are not generally being used for real-time, distributed applications, such as multi-party audio conferencing. Our primary goal was to determine whether a "real-time" audio application could be built using such a device and standard system software, such as UNIX and TCP/IP on an ethernet. We have accomplished this by implementing a simple audio conferencing program, and characterizing its performance with regard to delay and data loss.

Our second goal was to provide a convenient programming interface to the audio device. For this, we used the X Window System which provides a uniform interface for accessing both local and remote input and output devices. This allowed output to a remote audio device to be accomplished as simply as output to a remote window.

Beyond the interface, it is questionable whether it is appropriate for a window server to actually manage an audio device. Some have noted that audio devices should be managed by a networked "audio server," in

---

This research is supported in part by the National Science Foundation, Digital Equipment Corporation, NCR Corporation, and the Powell Foundation.

much the same way that screens are managed by window servers, but that this audio server should be completely separate from the window server [1]. As a first step, we decided to make simple modifications to the existing X window single server design. Thus, we present a very simple audio extension to X that hides the idiosyncrasies of the audio device from the programmer.

In the following section we discuss relevant past work. We present the design of the audio conferencing program in section 3, and our measured results in section 4. We finish with a few concluding remarks and comments on future directions.

2. Past Work

The complexity of a voice system is exemplified by the strict real-time requirements, large file sizes, and the devices needed for recording and playing audio files. The Etherphone System, developed at Xerox Palo Alto Research Center, was designed with features common to text file servers [7]. It includes features such as voice editing, sharing, and the use of voice in client applications, and is comprised of a voice file server, voice manager, and a voice control server. It was intended that the file server should have real-time requirements, though this had not yet been implemented. It does implement operations needed to allocate, record, and play voice files. The voice manager provides audio storage, and the control server manages interaction amongst Etherphones.

The VOX Audio Server was developed at Olivetti Research Center [1]. It differs from the Etherphone System as it stores the voice on the workstation where the server is running and does not have a centralized voice storage server. VOX supports flexible audio routing and sharing of audio resources between clients. Essentially, this system arbitrates resources between applications and provides device locking, mapping, and suspension and resumption. It applies the window server model to the management of audio devices.

Important differences between audio and graphics exist because audio is serial , requiring the server to switch between different speech modes and support interruptibility. Also, since audio implementations differ widely it is more difficult to provide hardware-independent software interface for audio and voice. Despite these differences, Binding et al. believe that the similarities are sufficient to base the audio interface on window-based interface. Their implementation of VOX is similar to that of a window server. It has layers of functionality with a device layer and a workstation agent layer. There is concurrency and a hierarchical composition. Also, though the resource management differs between the audio and window servers, they both provide resource allocation functions to applications. Finally, the VOX designers recognized the importance of synchronization between the VOX server and the window system (although it is not clear from the paper how this was accomplished).

ACME is a set of abstractions intended to extend a network window system to us e continuous media, such as video and audio [3]. Most continuous media activity is not handled by the actual window system, but by a separate ACME kernel which is implemented as another process or thread. The window system and ACME kernel communicate when necessary, after a window is moved, for example, when ACME must render video to the new location. Using ACME, a client application only sends requests to the window system, as usual, but there are new requests for accessing ACME resources. These requests are sent to ACME to be processed. The new requests are used to set up continuous media connections, separate network connections between the client and ACME kernel that audio or video data is sent over. In the ACME design continuous media is not mixed with graphics or window requests.

3. Design

We already had a text-based conferencing program that used X for distributing data to remote windows [6]. In order to integrate audio into this program, we implemented a simple extension to the X server to manage the audio device in the Sun SPARCstation. The application could then send text requests and audio requests using one mechanism, thus simplifying its job.

3.1. The Sun SPARCstation

The Sun SPARCstation plays and records a single channel of sound using the AM79C30A Digital Subscriber Controller chip. This chip has a built-in analog to digital converter (ADC) and a digital to analog converter (DAC). These converters drive the built-in speaker and an external headphone/microphone jack. Digital audio data is sampled at a rate of 8000 samples per second with 12-bit precision. The controller chip utilizes mu-law encoding to compress the 12-bit audio samples into 8-bit values. The sound quality of this audio data is equivalent to that of standard telephone service.

The audio controller chip is accessed as the audio device. Application programs can read audio data (record) by opening the audio device and initiating a *read()* system call. The device records audio data into an internal buffer, and the application's *read()* will block until the requested amount of audio data is available in the system buffer. After the first *read()* is issued, the chip samples continuously, and if this buffer overflows, sampling ceases until room becomes available. Sampling may be paused or stopped by the application directly using *ioctl()* calls.

The application can play audio with the *write()* system call. The data is written to the system buffer, which the chip reads and converts to sound. If the system buffer is full, the *write()* call will block until the transfer is completed. The *write()* returns when the data is successfully transferred to the system buffer; completion of audio output may take considerably longer.

3.2. The Audio Extension

The X protocol has a well-defined extension mechanism. On start-up, the X server calls the initialization routines of all extensions that were compiled into it. The initialization routine registers the extension name with the server, and also a dispatch routine that is called when the server receives one of the extension requests. This dispatch routine determines which of the extension routines to call to process the request.

We added the following X requests as part of our extension:

XOpenAudio(Display)

> Requests the X server to open the audio device, and set gain registers to default values.

XRecordAudio(display, buffer)

> char buffer[];
>
> Requests the X server to read audio data from the device and return it in the users buffer.

XPlayAudio(Display, buffer , sizeofbuffer)

> char buffer[];
>
> int sizeofbuffer;
>
> Requests the X server to play the audio data contained in the users buffer.

XSetAudioVolume(Display, playvolume, recordvolume)

> int playvolume, recordvolume;
>
> Requests the X server to set gain registers appropriately.

With the availability of these requests, our conferencing application needed only to issue an XAudioRecord to the local server (which recorded the local user's voice), and then issue an XAudioPlay to each of the remote X servers that are involved in the conference. Each of the other members of the conference must do the same. Thus, the text-based conferencing system needed approximately 10 lines added to it to implement audio conferencing. The connection topology of a three party conference is depicted in figure 3.1.

The PlayAudio request proved to be the only request that was difficult to implement. Since a *write()* to a full audio buffer will block until there is room to accept the transfer, it is possible for the X server to block

if it is flooded with PlayAudio requests (as would happen if a client dumps a file of audio to the server). Since this is intolerable, we decided that the X server should create a child process (Audio Manager) to watch over the audio device and feed it when necessary, while the server would only send the audio data to the child process via a pipe or shared memory. In this way, the X server can process the PlayAudio request quickly and respond to other requests. It is up to the child process to queue audio data if necessary. The relationship between conferencing application, X server, and Audio Manager is depicted in Figure 3.2.

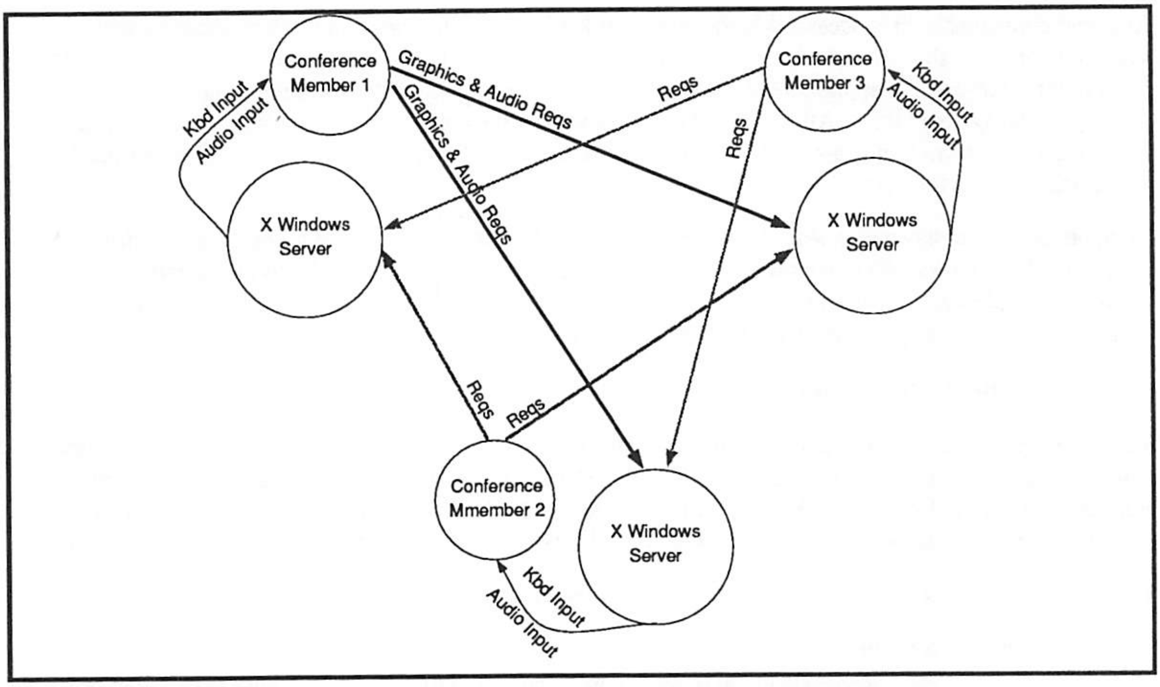


Figure 3.1: Connection topology of a three party conference. Each member requests keyboard events and audio input from its local X server, then sends graphics and audio requests to each of the other members' X server.

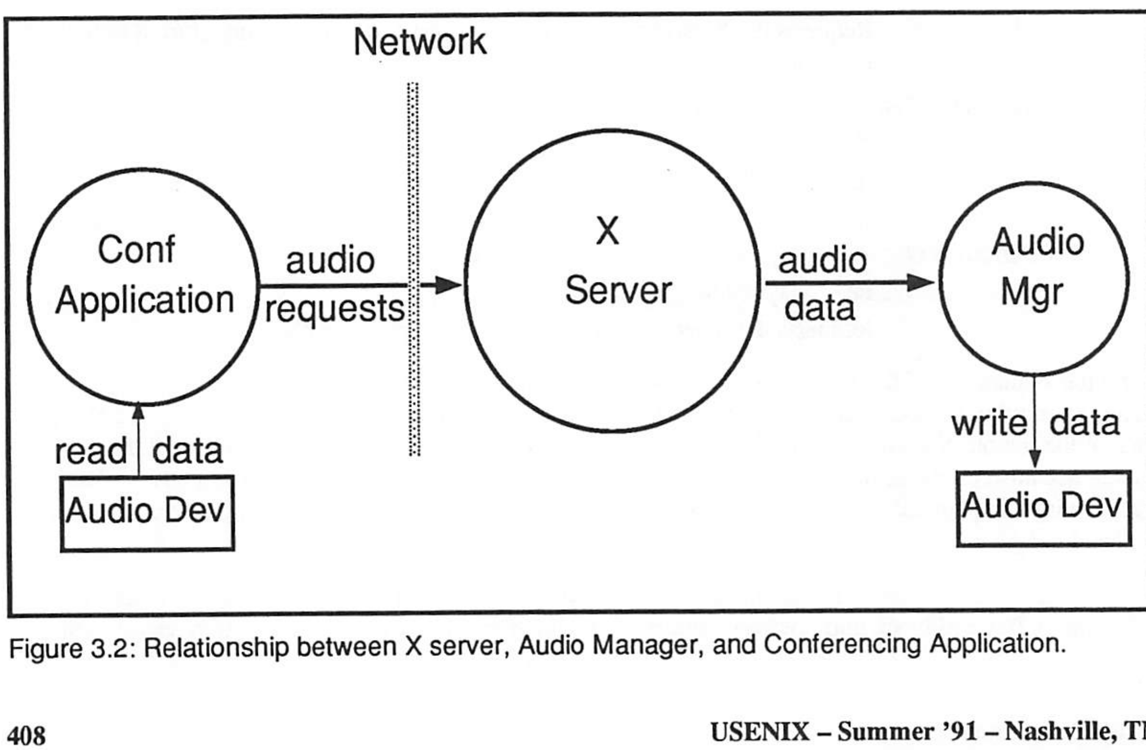


Figure 3.2: Relationship between X server, Audio Manager, and Conferencing Application.

4. Results

For applications that only want to play an audio file, it is advantageous for the client to read and send the audio data in large amounts. This causes a lot of audio data to queue up at the chip, which can then be played at a steady pace. However, if the conference application reads and writes large amounts of data, then a large amount of delay is introduced into the conversation (e.g., reading 8K from the audio device will block for 1 second). Thus, an appropriately small read/write size was used for time-dependent applications such as this. We will show what the optimal size is shortly.

Even with this modification, delay still crept into our conversations. When first started, delay was virtually nonexistent, but over the course of a five minute conversation two seconds of delay accumulated. Delay can be introduced into an existing conversation any time the audio device buffer empties. The amount of delay will be the length of time that elapses until the next *write()* occurs.

We hypothesize that the delay is introduced due to operating system overhead, e.g., when one of the processes (either client, X server, or Audio Manager), is switched out and is not able to service the audio queue. A real-time operating system is called for in this case, but we have devised a simple scheme to overcome our delay problem which works well enough for our voice-quality audio application.

4.1. Coping with Delay

Since delay is introduced into the conversation by allowing the audio queue to completely empty itself, and then writing the audio data that should have filled the hole to the queue, a strategy is needed to determine when to discard audio data. Several solutions have been suggested which we describe briefly [5].

> **Round Trip Measurement**: The sender places a timestamp in a packet and sends it to the receiver. The receiver returns the packet to the sender, which then computes the round trip delay of packet transmission, and sends this estimate to the receiver. All subsequent packets contain timing information relative to the first packet sent. Since the receiver has an estimate of network delay it can determine the estimated arrival time of a packet, and discard the packet if it arrives later than the estimate.
>
> **Absolute Timing**: A global clock must be maintained by all members of the conference, and all audio packets include a timestamp obtained from this clock. Any receiver can then examine the timestamp to determine if the packet's age is beyond a certain threshold; if so, it is discarded.
>
> **Added Variable Delay**: On long-haul networks, each intermediate node adds the time it has held the audio packet to a field in the packet header. Thus, any node along the route can examine this field to determine the packet's relative age. If the relative age exceeds a certain threshold, the packet is discarded.

Each of these strategies is somewhat cumbersome: global clocks are difficult to maintain; relative delay strategy needs to be implemented in system software. Our approach was to simply allow the Audio Manager to discard data if its internal queue held more than a certain amount of audio data.

We define BLOCKSIZE as the amount of audio data that is read from or written to the audio device buffer. When writing to the audio buffer, the Audio Manager reads everything from its internal queue; if the read amount, N, is a multiple of BLOCKSIZE (which it has to be, since the client sends audio data in units of BLOCKSIZE), only the most recent BLOCKSIZE data is written to the audio device queue, and the rest is discarded. The rationale behind this is that if more than BLOCKSIZE amount of data was able to accumulate, then the Audio Manager was not able to meet its deadline for the first (N-BLOCKSIZE) amount of data, so it shouldn't play it. One way to interpret this is that the Audio Manager has fallen behind, so it must somehow catch up.

We decided to implement this strategy on the record/send side of the conference as well. When reading from the audio device buffer, if the conference application finds more than BLOCKSIZE bytes in the buffer, it only sends the latest BLOCKSIZE bytes across the network to the X server. The accumulation of

extra bytes signifies that the application was not able to meet its deadline for reading the queue, so rather than leave the extra data there for the next *read()*, it is discarded.

4.2. Performance

We conducted a study to evaluate the performance of our system. The goal of this study was to determine whether audio could successfully be supported by a typical workstation running UNIX and X windows. Success was based on finding an optimal BLOCKSIZE which minimized delay and packet loss to an acceptable level, which we describe in more detail below.

In this preliminary study, we were not so much concerned with whether audio could be supported while the system was executing other user applications, but simply whether audio could be supported at all, given a quiescent system. By "quiescent system," we mean a system which does not have any additional user processes, but does have all the normal system-related daemons to support the normal operation of the system. Given the successful support of audio as part of the base line operation of the system, then one can experiment with methods of allowing user applications to execute with minimal interference to the audio, such as by always executing audio support processes at a higher priority than user applications.

In determining an optimal audio packet size, there is a trade-off between the delay and overhead: as the packet size increases, delay increases but overhead decreases. The overhead corresponds to the use of system resources devoted to supporting audio input, transmission, and output, which can have a negative impact on the performance of the rest of the system, and vice-versa.

The experimental setup comprised two Sun SPARCstations running the SunOS 4.0 operating system, one acting as a sender and one acting as a receiver, connected by a lightly loaded 10 Mbit/sec ethernet local area network. The X window client was located on the sender, and the relevant X window server was located on the receiver. Two sets of experiments were conducted, one under no-load conditions called the dedicated system, where absolutely no processes other than those related to our audio program were executing, and one under normal system-load conditions called the quiescent system, where the only other processes executing were system daemons. In both cases, there were no additional user processes and no additional user network activity other than what was due to system-related daemons. Thus, our results tell us the best we can expect given a completely dedicated system (which is only used for comparison since it is unreasonable to expect a workstation to be completely dedicated to audio), and in a quiescent SPARCstation environment with all system daemons still in force.

The main events of interest were the sending, receiving, and discarding of audio packets. Thus, we measured the time of every packet sent and received, and the amount of data discarded on a send or receive. This was measured at user level by calling *gettimeofday()*, which produced a time with one millisecond resolution. Times and amounts of discarded data were recorded to memory, and were stored on disk after the entire experiment. The problem of delays between the occurrence of events and the reading of the time were dealt with by collecting large numbers of samples and using statistical techniques.

The two experiments each consisted of twelve measurement sessions, each lasting fifteen minutes. The BLOCKSIZE was varied from 64 bytes to 2048 bytes, doubling the size between sessions. This produced six different sessions per experiment; these were each repeated again to obtain statistical significance for our measurements.

4.2.1. Dedicated System Experiments

When our system was completely dedicated to supporting audio, we observed that for all packet sizes the system executed almost perfectly. There was very little variance in send and receive rates, and the fraction of lost packets was very low. From this, we can conclude that voice-quality audio can certainly be supported if a machine is completely dedicated to this service. Of course, this is unreasonable to expect, but at least we know that it is possible, and that any deterioration under any loading conditions is due to the addition of load.

4.2.2. Quiescent System

We now consider the experiment under a quiescent system. Here we will see the effects of system processes on various metrics of audio performance.

4.2.2.1. Send and Receive Rates

Table 4.1 contains inter-send time statistics for the different packet sizes. For each BLOCKSIZE, it shows the percentiles in steps of 25, the mean, and the standard deviation.

Since the audio sampling rate is 8000 bytes/sec, the mean inter-send time is s imply the packet size divided by 8. The sample means in table 4.1 correspond closely to this formula. Notice that the standard deviation is small (approximately 2) and fairly constant for packet sizes between 64 and 1024, but almost doubles (to 4) for 2048. This tells us that the periodic sending of packets occurs with little variation.

Client Inter Send Time Statistics

BLOCKSIZE	0% (MIN)	25%	50% (MEDIAN)	75%	100% (MAX)	Mean	Stdev
64	0	8	8	8	341	8.02	1.39
128	0	16	16	16	272	16.03	1.90
256	0	32	32	32	135	32.10	1.45
512	0	64	64	64	220	64.01	2.25
1024	0	128	128	128	257	127.99	2.01
2048	0	256	256	256	300	255.92	3.97

Table 4.1: Client Inter Send Time Statistics.

Table 4.2 contains inter-receive time statistics for the different packet sizes. Like the mean inter-send time, the mean inter-receive time is the packet size divided by 8, and the sample means in table 4.2 correspond closely to this formula. However, the variation is significantly higher than that of the inter-send times, increasing logarithmically with the packet size. The progressive increase is to be expected, since the total variation observed at the receiver can result from variations due to sending, transmitting, and receiving the packet. The variation is still very small relative to the mean inter-receive time, as can be seen by the almost nonexistent differences in the 25th, 50th, and 75th percentiles. The non-zero values for the standard deviation are mainly due to a small number of outliers which have large values (note the difference between the mean values and the maximum value, given by the 100th percentile).

For both the inter-send and inter-receive times, variation is almost nonexistent for at least 50% of the samples about the mean. This is illustrated by the equality of the 25% and 75% percentiles for almost all statistics shown in tables 4.1 and 4.2, except for the inter-receive time with 1024 byte packets, where the difference is 2 milliseconds. Thus, most of what is contributing to the standard deviation is due to outliers, high inter-send or inter-receive times which are not common.

This tells us that under quiet conditions, there is very low variation in sending and receiving rates. However, significant variations can occur. To investigate this further, we looked at the time-series of inter-send and inter-receive times for each of the packet sizes, whose graphs appear in figures 4.1 through 4.12. Consider Figure 4.1, which contains the client inter-send time-series for a packet size of 64 bytes. We see that most of the inter-send times are approximately 8 milliseconds, which is the mean inter-send time. We also see that periodically, every 30 seconds, the inter-send time is near 50 milliseconds. This is due to

Client Intersend Time, 64

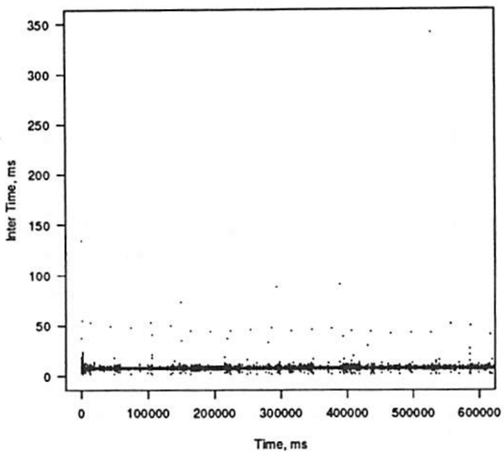


Figure 4.1

Client Intersend Time, 128

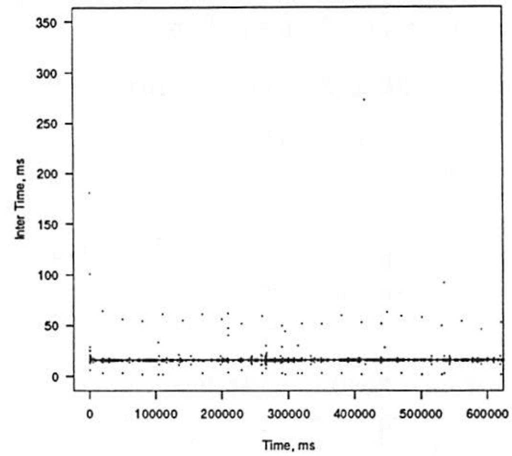


Figure 4.2

Client Intersend Time, 256

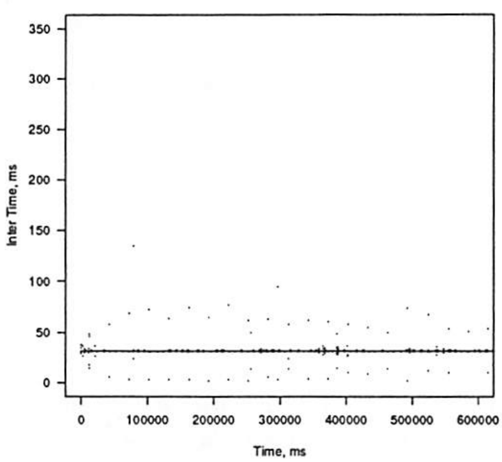


Figure 4.3

Client Intersend Time, 512

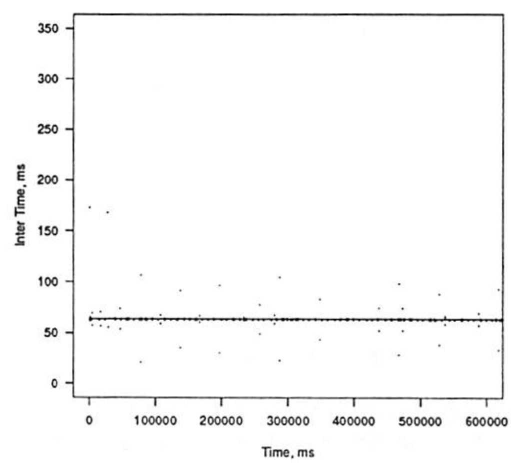


Figure 4.4

Client Intersend Time, 1024

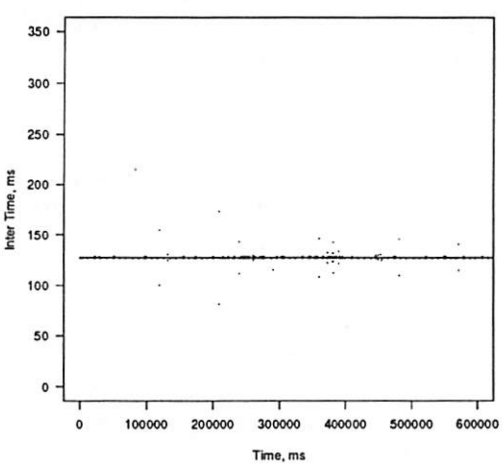


Figure 4.5

Client Intersend Time, 2048

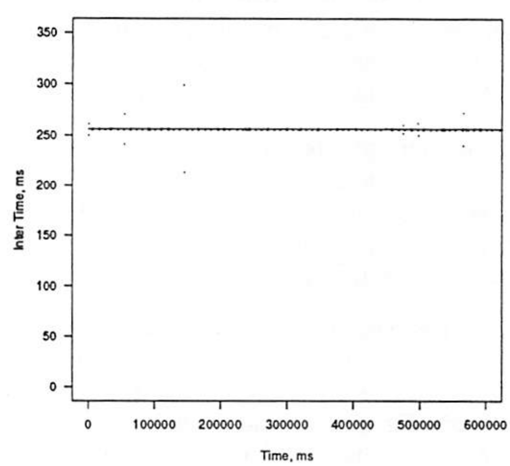


Figure 4.6

Server Intersend Time, 64

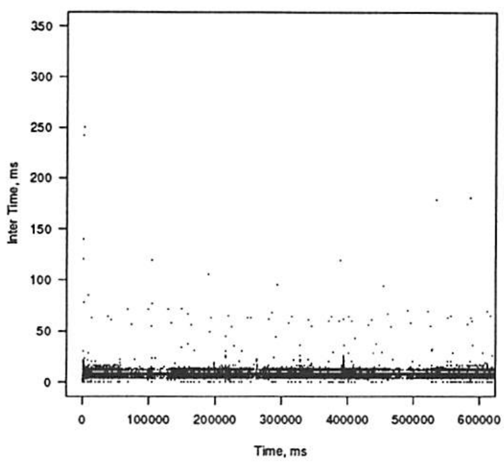


Figure 4.7

Server Intersend Time, 128

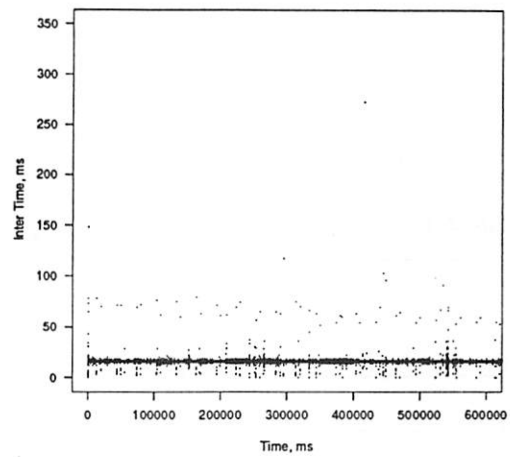


Figure 4.8

Server Intersend Time, 256

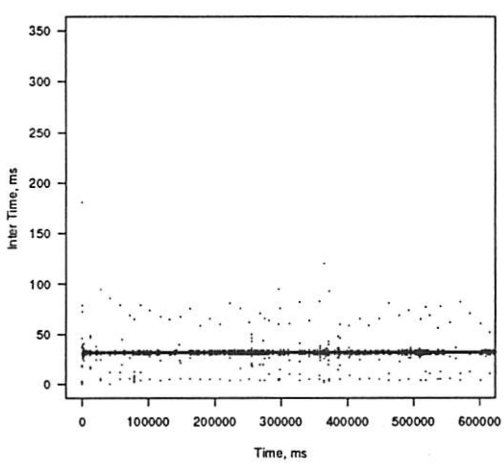


Figure 4.9

Server Intersend Time, 512

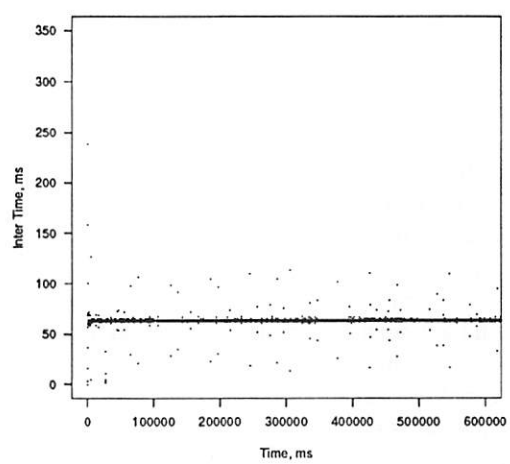


Figure 4.10

Server Intersend Time, 1024

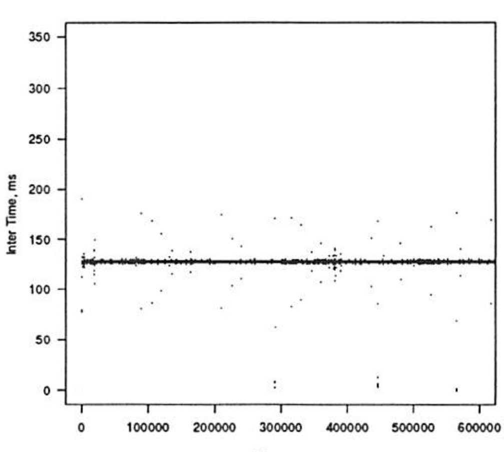


Figure 4.11

Server Intersend Time, 2048

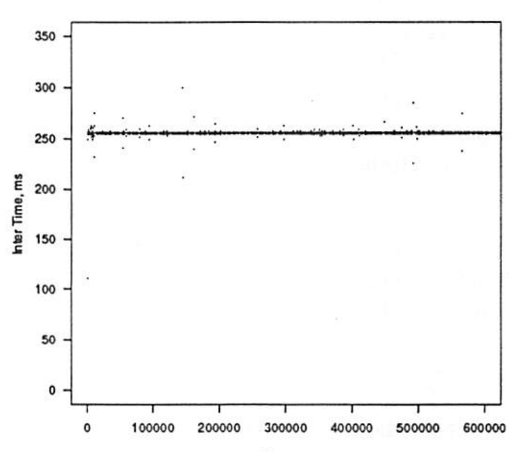


Figure 4.12

a system-related daemon which executes every 30 seconds, delaying the send process from obtaining the CPU. When the send process is switched in, it immediately sends the latest audio packet (the others are discarded), and since the next packet will be ready in 4 milliseconds, there is a pattern of inter-send times equal to 4 every 30 seconds. We have estimated that the daemon consumes approximately 42 milliseconds of CPU time; thus, depending on when the daemon is switched in, the inter-send time during this period can fluctuate between 48 milliseconds and 56 milliseconds.

Server Inter Receive Time Statistics

BLOCKSIZE	0% (MIN)	25%	50% (MEDIAN)	75%	100% (MAX)	Mean	Stdev
64	0	8	8	8	250	8.01	2.85
128	0	16	16	16	272	16.02	2.59
256	0	32	32	32	181	31.97	3.24
512	0	64	64	64	239	63.89	4.29
1024	0	127	128	129	263	127.85	5.81
2048	0	256	256	256	309	255.75	7.56

Table 4.2: Server Inter Receive Time Statistics.

The two patterns of high and low inter-send times occurring every 30 seconds is more clearly observable in Figure 4.2, where the packet size is 128. The average inter-send time is 16 as expected; the sequence of high inter-send times has a mean of approximately 55 milliseconds, and the sequence of low one has a mean of approximately 9 milliseconds (the total must be a multiple of 16). This is again observed in Figure 4.3, where the packet size is 256. However, in Figures 4.4 through 4.6, we see a different pattern in the high and low inter-send times: corresponding low and high inter-send times are symmetric about the mean inter-send time. For instance, in Figure 4.4 where the packet size is 512, the mean inter-send time is 64 milliseconds. Every 30 seconds, we see a pair of inter-send times which are equidistant from the 64 milliseconds. This is because the mean inter-send time for packet sizes 512 through 2048 is greater than the 42 milliseconds consumed by the daemon process. The same patterns can be observed in the inter-receive time-series, shown in Figures 4.7 through 4.12. From both the inter-send and inter-receive time-series, we see that most of the variation in inter-send times is due to the periodic execution of unrelated processes, which interfere with the audio process.

4.2.2.2. Delay

Table 4.3 contains sender-to-receiver packet delay statistics. The mean delay ranges from 3.7 milliseconds for small packet sizes to 7.2 milliseconds for the largest packet size. Mean delay and packet size are related by the following formula: $\text{delay}(s) \approx 3.5 + .002s$ milliseconds.

Thus, there is a constant delay of 3.5 milliseconds due to transmitting a packet of any size, plus an added 2 microseconds for every byte sent. Notice that the standard deviation decreases as the packet size increases, although there is a major decrease when the packet size increase from 64 to 128 bytes, and then only a slight decrease for larger packet sizes.

To determine the source of variation, we look at the time-series of packet delay. These are shown in Figures 4.13 through 4.18. As in the case of inter-send and inter-receive times, the main source of variation is due to the periodic execution of the 30 second daemon process. Every 30 seconds, delays ranging from the mean delay up to 50 milliseconds are experienced. The density of packets delayed greater than the mean decreases significantly with packet size; for packet sizes of 512 and greater, the relative number of

Packet Delay, 64

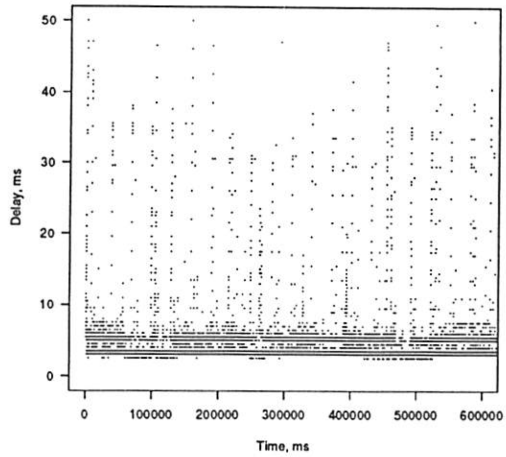


Figure 4.13

Packet Delay, 128

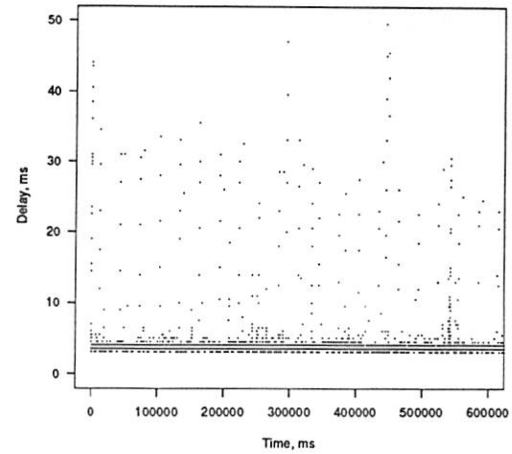


Figure 4.14

Packet Delay, 256

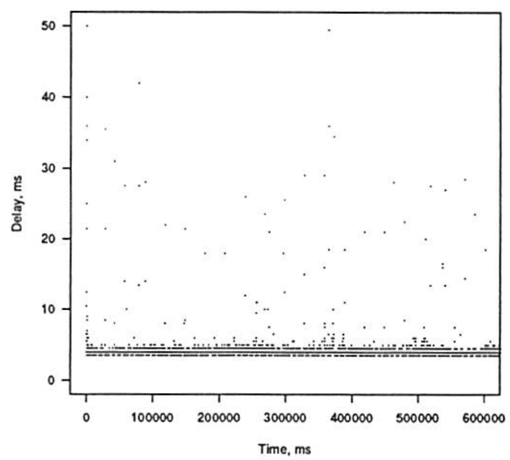


Figure 4.15

Packet Delay, 512

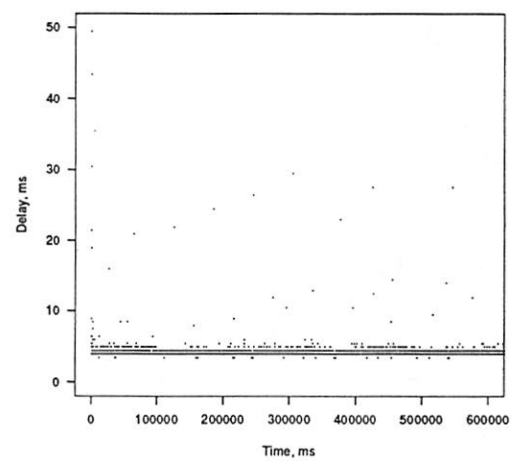


Figure 4.16

Packet Delay, 1024

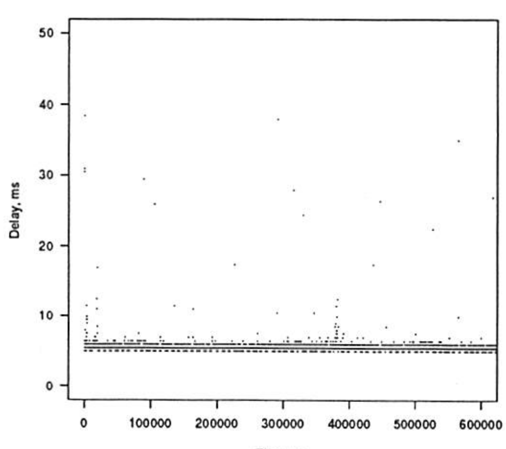


Figure 4.17

Packet Delay, 2048

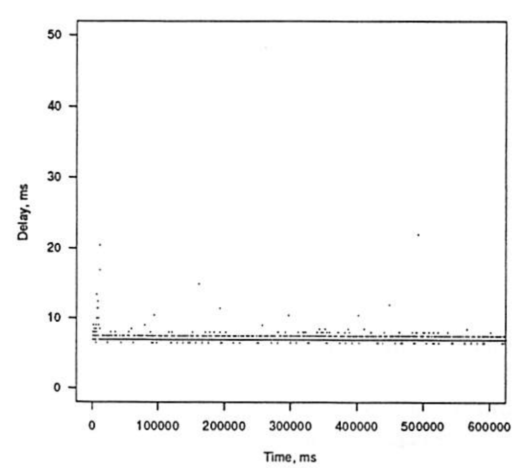


Figure 4.18

packets that are delayed for more than the mean value is very small.

Packet Delay Statistics

BLOCKSIZE	0% (MIN)	25%	50% (MEDIAN)	75%	100% (MAX)	Mean	Stdev
64	2.5	3.0	3.0	3.5	50.0	3.71	2.06
128	3.0	3.5	3.5	3.5	49.5	3.67	1.27
256	3.5	4.0	4.0	4.0	50.0	3.99	1.20
512	3.5	4.0	4.0	4.5	49.5	4.29	1.07
1024	5.0	5.5	5.5	6.0	38.5	5.65	1.12
2048	6.5	7.0	7.0	7.5	34.0	7.22	0.80

Table 4.3: Packet Delay Statistics.

4.2.2.3. Efficiency

Finally, we consider the efficiency of the system for different packet sizes, where efficiency is defined as the ratio between amount of audio data output at the receiver and the amount of audio data input at the sender. Table 4.4 contains the total amount of data discarded at the sender and at the receiver averaged over each of the 10 minute measurement sessions, along with the efficiency rating, for each packet size. We see that the amount of data discarded goes to zero for packet sizes 512 or greater.

Discarded Data Statistics

BLOCKSIZE	SENDER	RECEIVER	% EFFICIENCY
64	24084	25088	98.98
128	12112	2048	99.71
256	1560	1280	99.94
512	172	0	100.00
1024	0	0	100.00
2048	0	0	100.00

Table 4.4: Data Discarded Statistics

4.2.2.4. Best Packet Size

Given the analysis as presented above, the best packet size is 512 bytes. This size caused an inherent delay of 64 milliseconds (time to gather the data) plus an end-to-end average delay of 4.3 milliseconds for a total of 68.3 milliseconds, with a standard deviation of 1.1 milliseconds. For real-time conversations, this amount of delay is acceptable[4]. Furthermore, variation in inter-send time and inter-receive time statistics is relatively low (in particular, the variation jumps significantly for inter-receive times for packet sizes above 512). Finally, we note that the efficiency is virtually perfect for 512 byte packets, but decreases for

smaller sizes.

5. Summary and Future Work

We have shown that it is possible to modify the X server to service audio requests and attain acceptable performance if the user only uses the audio application. This set of experiments demonstrates that audio can be done within X, and that any problems with the existing system can be corrected with better implementations of the server.

It is true that sending many X requests to a server may keep it so busy that it can not service audio requests in a timely manner. However, this is only a problem with the existing implementation of X Windows. Specifically, it is a single process that services all client connections in a round robin manner. Furthermore, it completes each request in its entirety before reading another. Thus, with many clients requesting service a client that makes audio requests may receive poor service (and in our scheme, much audio data would be discarded).

To overcome this problem, the X server must be implemented as a collection of cooperating threads sharing a single address space [2]. Variations of this idea can insure that one client does not starve another, and that audio requests do not wait in a service queue needlessly. This also simplifies the implementation of synchronization between different types of media, since event queues can be shared. It simplifies the implementation of a video thread as well, since that thread would have access to all window system data structures, such as clip lists, that it needs to render properly. With separate processes, some sort of server-to-server communication would be needed to insure this.

6. Acknowledgments

We gratefully acknowledge the help received from Barbara Bittel, Vach Kompella, Jim Mattson, Keith Muller, Rick Ord, and Dipti Ranganathan.

References

1. B. Arons, C. Binding, K. Lantz and C. Schmandt, "The VOX Audio Server", Multimedia '89: 2nd IEEE COMSOC International Multimedia Communications Workshop, Ottawa, Ontario, April 20-23, 1989.

2. J. Brezak, I. Elliot and N. Myers, "A Multi-threaded Server for X and PEX", Proc. X Technical Conference, 1990, Boston, MA, Jan 1990.

3. G. Homsy, R. Govindan and D. P. Anderson, "Implementation Issues For A Network Continuous-Media I/O Server", Technical Report No. UCB/CSD 90/597, Computer Science Div., EECS Dept., Univ. of Calif. at Berkeley, September 1990.

4. "Notes on the Network", American Telephone and Telegraph Company, Network Planning Div., Fundamental Network Planning Section, 1980.

5. W. A. Montgomery, "Techniques for Packet Voice Synchronization", IEEE Journal on Select Areas in Communications Vol. SAC-1 No. 6 pp. 1022 - 1028, December, 1983.

6. J. Pasquale, E. Anderson, M. Bubien, K. Fall, V. Kompella, S. McMullan, D . Ranganathan, R. Terek, "Operating system and window system research for multi-media applications: a status report", U.C. San Diego Technical Report CS90-176, July 1990.

7. D. B. Terry and D. C. Swinehart, "Managing Stored Voice in the Etherphone System", Trans. Computer Systems 6, 1 (Feb. 1988), 3-27.

Robert Terek is a graduate student at the University of California, San Diego,where he will be completing the MS degree in Computer Science in June 1991. He received the BA degree in Computer Science from Boston University in May, 1986. His research interests include computer graphics, window system design ,and multimedia computing systems. Prior to entering graduate school, Robert worked at both Sun Microsystems and Apollo Computer on window system software implementation.

Joseph Pasquale is Assistant Professor of Computer Science and Engineering at the University of California, San Diego, and Senior Fellow at the San Diego Supercomputer Center. He received the PhD degree in CS from UC Berkeley in 1988, and the BS and MS degrees in EECS from MIT in 1982. In 1989, Dr. Pasquale received the National Science Foundation Presidential Young Investigator Award. Dr. Pasquale has been a computer systems consultant for a large number of industrial and governmental research organizations, including AT&T Bell Laboratories, Bell Communications Research, Digital Equipment Corporation, NCR Corporation, and the National Security Agency. His research interests include distributed systems, multimedia computing systems, operating systems, and performance evaluation.

Integrating Audio and Telephony in a Distributed Workstation Environment

Susan Angebranndt (susan@wsl.pa.dec.com)
Richard L. Hyde (rich@wsl.pa.dec.com)
Daphne Huetu Luong (luong@wsl.pa.dec.com)
Nagendra Siravara (siravara@wsl.pa.dec.com)
Digital Equipment Corporation

Chris Schmandt (geek@media-lab.media.mit.edu)
MIT Media Lab

Abstract

More and more vendors are adding audio, and occasionally telephony, to their workstations. At the same time, with the growing popularity of window systems and mice, workstation applications are becoming more interactive and graphical. Audio provides a new dimension of interaction for the user, and the possibility of a powerful new data type for multi-media applications.

This paper describes our architecture for the integration of audio and telephony into a graphics workstation environment. Our framework is a client-server model; at the heart is an audio server that allows shared access to audio hardware, provides synchronization primitives to be used with other media and presents a device-independent abstraction to the application programmer.

1. Desktop Audio

This paper describes a server designed to provide the underlying audio processing capabilities required by families of workstation-based audio applications. It is an essential component of the concept of *desktop audio*, a unified view encompassing the technologies, applications, and user interfaces to support audio processing at the workstation. This section describes the technologies underlying desktop audio, and some example applications. In addition we discuss requirements of both the applications and their user interfaces, because these dictate some of the features and performance required of an audio server.

1.1. Technologies

The basic technology for manipulation of stored voice in a workstation is *digitization*, which allows for recording and playback of analog speech signals from computer memory. At one extreme, telephone quality recording requires 8,000 bytes per second; at the other extreme the quality of a stereo compact audio disc consumes just over 175,000 bytes per second. The low end of this scale is easily within the performance of current workstations, and basic digitization is becoming commonplace. Some workstations already support CD quality coding, and this, too, will become universal within the next several years.

Text-to-speech synthesis allows computers to convert text to a digital speech signal for playback. Synthesis is usually broken into two processing steps. The first step converts the text to phonetic units; although a linguistically difficult task, this is most easily implemented on a general purpose processor. The second step is a vocal tract model capable of generating an appropriate waveform from the units generated by the first step; this has traditionally been performed on a digital signal processor.

Speech recognition allows the computer to identify words from speech. Speech recognition usually employs a digital signal processor to extract acoustically significant features from the audio signal, and a general purpose processor for pattern matching to determine which word was spoken. Although there is much talk about the ''listening typewriter'' which can convert fluent speech to a text document, this is well beyond the capabilities of currently available recognizers, which have small vocabularies and require both a careful speaking style as well as head-mounted microphones or an acoustically controlled environment.

The *telephone* can be thought of as a voice peripheral, just like a loudspeaker, and is a key component to desktop audio. Voice messages and applications for remote telephone-based workstation access will likely be a primary source of stored voice used as a data type. A small amount of electronic circuitry can provide an interface to analog telephones. ISDN, the international standard for digital telephony, is driven by a data communication protocol which can be easily managed by modern workstations.

Until recently, most audio devices required special purpose hardware, resulting in the associated high cost of audio systems. But with faster workstations and plentiful memory, more and more audio processing can be implemented on the workstation itself, with little or no special hardware. The real-time requirements of managing a stream of 64 kilobit per second voice or the even slower data link of an ISDN telephone connection are well within the capabilities of existing workstation platforms. Many speech processing techniques which have traditionally been implemented on DSPs are now within the capabilities of general purpose microprocessors. The upshot of these developments is that audio is about to become universal, provided that adequate applications with well-designed interfaces can be made available to users.

1.2. Applications

Audio processing employing the technologies enumerated above will be used by a variety of applications. They will take advantage of various attributes of voice: its richness, its primacy in human communication, the ease with which we transmit it over a distance by telephone, and our ability to speak and listen while performing other, non-audio tasks.

With the ability to control the telephone, a workstation can be used to place calls from graphical speed dialers, an address book, or telephone-based "dial by name" (which allows the caller to enter a name with touch tones). Workstation-based personal voice mail allows graphic display and interaction with voice messages, and can provide the ability to move messages to other voice-capable applications, such as an appointment calendar. Voice and text messages can be merged into applications that provide for screen or telephone access to each.

Because it is rich and expressive, voice can be very useful for annotating text, such as the marginal notes on a document under review or as a quick header to a forwarded message in some other medium. Stored voice can be used more formally as an essential component of multi-media presentations.

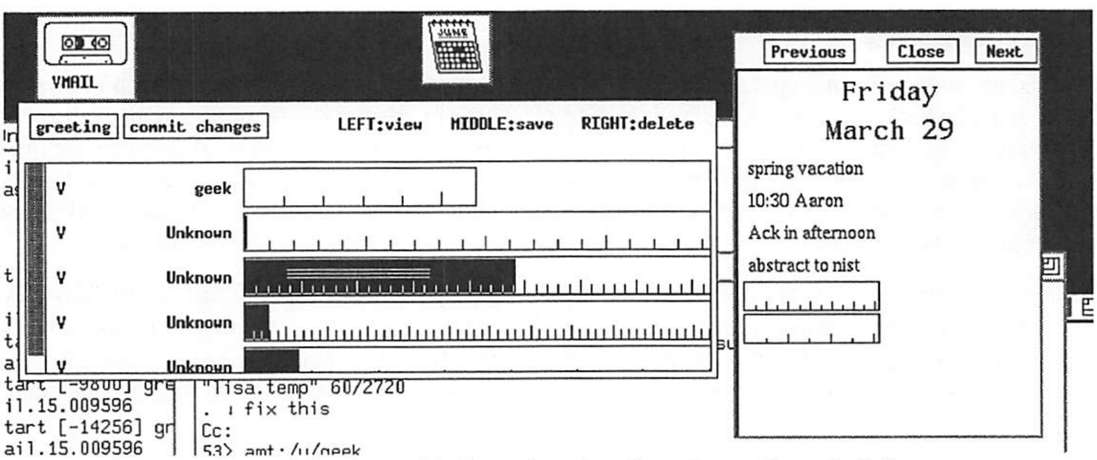


Figure 1-1: A graphical user interface for voice mail, on the left, allows telephone messages to be moved to a users calendar, on the right, in two applications developed at the MIT Media Lab.

Speech synthesis and recognition allow for remote, telephone-based access to information accessible by the workstation. Speech recognition can increase user performance for hand-and-eye busy applications such as CAD and management of the window system. Synthesized speech or playback of distinctive

sounds can be much more effective for alerting than the universal "beep" employed in UNIX[1] applications such a "biff", "talk", "wall", editors, and broadcast system messages.

1.3. Application requirements

As inviting as these applications may sound, in general no single application can justify exclusive use of the workstation audio hardware. Instead, we envision a wide variety of applications, of which those just mentioned, all running in concert. These applications will make use of audio as a data type, and the user must be able to move audio between applications and transmit it between sites.

This is really not very different from current uses of text; a user may invoke an editor while sending electronic mail, or copy a portion of the text displayed in one window into an application running in another. For an example in the audio world, consider Figure 1-1, which shows window-based graphical user interfaces for copying sound between applications.

The various audio applications will be designed independently and run as separate processes. But they must share limited resources, such as speakers and microphones, just as window applications share screen pixels and the mouse. These applications need to communicate among themselves to coordinate the sharing of data and allow for more powerful management of the applications by the end user; such a communication mechanism must support inter-process message interchange.

1.4. User interface requirements

Although all these applications can make powerful use of audio, the medium is intrinsically difficult to employ in computer applications. Stored voice is awkward to handle since we cannot yet perform keyword searches on it. It is slow to listen to (although fast to create), and is a serial medium because of its time-dependent nature. A loudspeaker broadcasts throughout a room, disturbing others in the area and allowing them to hear possibly personal messages. Finally, the technologies themselves are currently limited; synthesized speech is difficult for unaccustomed listeners to understand, and speech recognition simply does not work very well.

Because of these limitations, graphical interaction styles will dominate when a screen is available. Graphical representations, such as the SoundViewer widgets depicted in Figure 1-1, can provide both visual cues to the duration of the sound as well as a means of interacting with playback and navigating within the sound.

Both audio playback and interactive audio systems have stringent real-time requirements. Although playback of stored voice may not require a high data bandwidth within the workstation, once playback starts it must continue without the slightest interruption; this is very different from refreshing a static text or graphic display. Since voice recognition and even touch tone decoding are quite error prone, the user needs immediate feedback that input has been recognized. In a voice-only interface this is essential but may be difficult.

Because of the difficulties building effective audio interfaces and applications, users have yet to realize its full potential. Many applications written to date have employed weak interfaces; interfaces must be designed with full appreciation of the limitations in the voice channel. For the near future we can expect, and must encourage, a great deal of experimentation in the design of audio user interfaces, both screen- and telephone-based.

2. Requirements of desktop audio

The previous section described the world of desktop audio. This view assumes access to a variety of audio processing technologies, largely implemented as software, from a number of applications running simultaneously, with demanding user interface requirements. What is needed from a software architecture to support such applications?

[1]UNIX is a trademark of AT&T Ball Laboratories

The primary need is *resource arbitration.* Applications should be written without having to worry about mechanisms to share resources with other applications (although they will have to have a strategy to deal with not gaining access to a critical resource).

Another aspect of resource management is *sharing* and allowing multiple applications to use the audio hardware. For instance, the multiplexing of output requests from a number of applications to a single speaker, to be heard simultaneously. Or distributing words detected by a speech recognizer to the proper applications.

It is also important that the software interface be *device independent* regardless of actual workstation hardware. Device independence provides for portability, which encourages application developers, and allows workstation vendors to introduce more powerful hardware devices without abandoning previously written applications.

This interface should also provide applications with *networked access* to resources. This promotes sharing of applications and data, and allows a user to more easily access applications when not at his or her own workstation. Additionally, networked access allows many workstations to share critical or expensive resources which cannot easily be replicated on every desk.

The software underlying desktop audio applications must support the *real-time* requirements of maintaining an uninterrupted stream of audio data once playback starts. Quality user interactions demand the ability to start and stop audio playback quickly, and deliver input events to applications with little latency. Audio operations must be *synchronized* to support seemless playback of multiple sounds in sequence, or to quickly transition into record mode after playback of a voice prompt while taking a message. Audio operations must also be be synchronized with other media, to support both multi-media presentations as well as the use of graphical user interfaces that control audio playback.

Because audio can be stored using a variety of encoding methods, it is useful to support *multiple data representations* at a level below the application. This is important for several reasons. First, over the next several years users will demand higher quality speech coding and workstations will become fast enough to support this; if every application must be rewritten progress will be delayed. At the same time, improved compression techniques will be used to transmit voice across local and wide area networks, reducing bandwidth at the price of data representation complexity. Applications should be sheltered from this.

Finally, audio support should be *extensible* to support new devices and signal processing algorithms as they emerge. Our approach is to provide a device subclassing mechanism in the server, allowing extension of the class hierarchy using existing protocol capabilities.

3. Why a client - server model?

In order to satisfy the requirements of the types of applications and technologies described above, we have built an audio server. A server, running as a single, separate process to support a number of clients (applications) simultaneously, is advantageous for several reasons. A server approach has been utilized successfully in the past, across a wide range of applications, technologies, and operating systems [2], [1], [4].[2] The server concept has been widely accepted for window systems, and in fact we can apply much of what we have learned from our experience with the X Window System[3] [3] to the audio domain.

A server is required for resource sharing and arbitration across multiple applications; there must be some point at which all application requests meet so that resource contention can be arbitrated. Use of a well-defined protocol for communication with the server allows the application to remain compatible across multiple vendors' server implementations, and its device-independent nature shelters the application from hardware differences.

---

[2]The Olivetti VOX audio server introduced the concepts of typed server-side entities (corresponding to devices) and application compositing of devices, and also borrowed heavily from X. There are a number of similarities between VOX and our audio protocol. The design of the prototype VOX server was more oriented towards control of analog audio devices.

[3]X Window System is a trademark of The Massachusetts Institute of Technology.

Use of a separate process for control of a real-time medium such as stored voice simplifies application design in many respects. Audio playback requires repeated calls to buffer management routines to move digital audio data from disk storage to audio hardware. Audio streams may be mixed, or effects added, by introducing more buffers and operators upon them. Continual feeding of buffers is a distraction for application software, which is more easily written and maintained as a set of routines, each responding to some user stimulus or server-generated event such as sound playback completion.

Although many parts of the audio server internals can be modeled after the X server, it is important to keep the two separate. The server for a window system has very different real-time requirements than an audio server; they cannot be merged without serious design compromises for both. Adding audio to a graphics server adds needless complexity to components supporting each medium. Telephone-based audio applications may well run on a workstation not running any window system, and should not be burdened by many lines of display support code.

The strongest argument for merging media into a single server is *synchronization* between media. A single server can provide internal methods of controlling one medium as a function of progress though another, which is useful if the media have varying latencies or throughput characteristics. This is a minor problem; most synchronization will happen in response to some user input rather than the internal state of the server. This entails round trips for messages between client and server, at which point it is much less critical whether there is a single server or many.

In fact, servers for multiple time-dependent media such as audio and video will almost certainly employ a multi-threaded architecture internally, with all the associated problems of state and communication between threads. So the cost of multiple servers for synchronization can be reduced to the cost of the context switch between server processes and data sharing across server address spaces. With vastly improving processor speeds and increased hardware support for context switches, these differences are probably minor.

4. System model

Our audio architecture consists of five main components: the audio network or *protocol*, an audio server that implements the protocol, a client-side library (Alib), a user-level toolkit, and applications. The relationship between the components is shown in Figure 4-1. The structure of each of these components is briefly described in the following sections, with the remainder of the paper focusing on the protocol, the server implementation, and communication with the server.

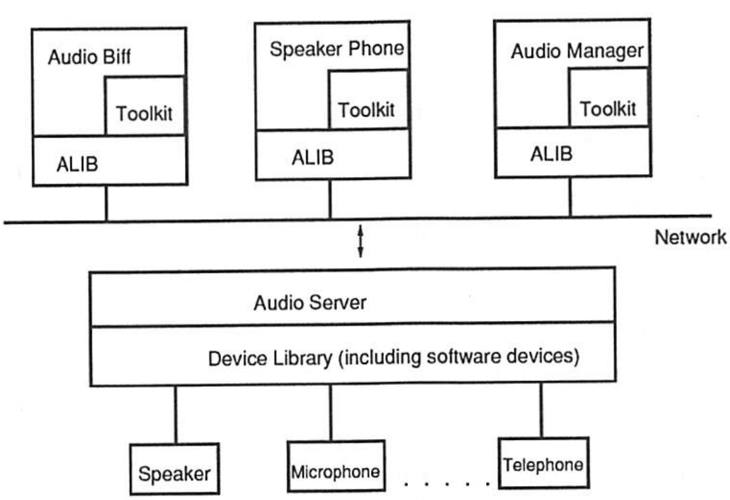


Figure 4-1: Audio System Model

4.1. The Audio Server and the Audio Protocol

For each workstation, there is a controlling server. The server implements the requests defined in the protocol and executes on the workstation where the audio hardware is located, providing low-level functions to access that hardware and coordination between applications. Clients and a server communicate over a reliable full duplex, 8-bit byte stream. A simple protocol is layered on top of this stream. The audio server can service multiple client connections simultaneously, and a client can have multiple connections to one or more audio servers. This protocol is a very precisely defined interface. The tight definition of the semantics of the protocol make it independent of the operating system, network transport technology and programming language. The ''Alib'' library provides clients with a procedural interface to send and parse the protocol messages.

Requests are asynchronous, so that an application can send requests without waiting for the completion of previous requests. Some requests do have return values (state queries, for instance), which the server handles by generating a reply which is then sent back to the application. The client-side library implementation can block on these requests or handle them asynchronously. Blocking on a request with a reply is tantamount to synchronizing with the server. Errors are also generated asynchronously, and applications must be prepared to process them at arbitrary times after the erroneous request.

The audio protocol describes several major pieces :

1. *connections* that provide the communication between server and client
2. *virtual devices* that specify a device-independent abstraction of the actual hardware. These are combined to build audio entities that can play, record or otherwise interact with the user.
3. *events* that notify the client of changes in state (for instance that a play command has completed).
4. *command queues* that control the use of the virtual devices and provide synchronization among devices and commands.
5. *sounds*, or audio data repositories, that can be played or recorded.

4.2. Alib and the Toolkit

Alib is simply a procedural interface to the audio protocol. It is a ''veneer'' over the protocol and is the lowest level interface that applications will expect to use. Applications should not use the workstation hardware interface directly or bypass the library.

We have built a toolkit that sits on top of Alib. The goals of the toolkit are to: hide or automate wiring of devices for greater portability, hide the location and format of sound data, hide and manage device queue management, and provide mechanisms for synchronizing audio with other media (for example, X graphics). Clients use the toolkit to construct audio user interfaces, such as an audio dialogue or touch tone-based menu. However, the toolkit is ''policy free'' in that it does not enforce a particular style but attempts to provide a mechanism for interaction. Further discussion of the toolkit is beyond the scope of this paper.

4.3. The Audio Manager Client

In a window system, a special application called a ''window manager'' mediates the competing demands for scarce resources such as screen space, input focus, and color selection. The window manager sets the policy regarding the input focus of the pointing device and keyboard; it keeps track of windows and sets the policy for moving and resizing them. Because the audio protocol allows multiple clients to access the audio hardware simultaneously, an application similar to a window manager is needed to enforce contention policy. We call this the *audio manager*.

5. Protocol Overview

5.1. Audio device abstraction

The protocol provides applications with a device-independent interface to the audio capabilities of the workstation. The device-independent objects, *virtual devices*, are the basic building blocks of "audio structures" in the protocol. Each virtual device can be described by a class name, a set of attributes, a set of controls, and a set of device ports. Device ports represent audio inputs and outputs, known respectively as *sink* ports and *source* ports. The ports are used to connect virtual devices together and define the audio data path between them.

Audio structures are constructed by organizing one or more virtual devices within containers called *logical audio devices* or *LOUDs*. LOUDs can then be constructed into a tree hierarchy. This hierarchy is used to logically group virtual devices into manageable substructures, such as a tape recorder that plays and records, or an answering machine.

Once a LOUD tree is built, two or more virtual devices can be combined to create more complex device abstractions by connecting *wires* between them. The wires specify the flow of data between the devices. The root of the LOUD tree is used to control and coordinate the audio streams to the LOUDs in the tree. A command queue is provided for each root LOUD. Figure 5-1 shows the hierarchy and wiring for an answering machine LOUD tree.

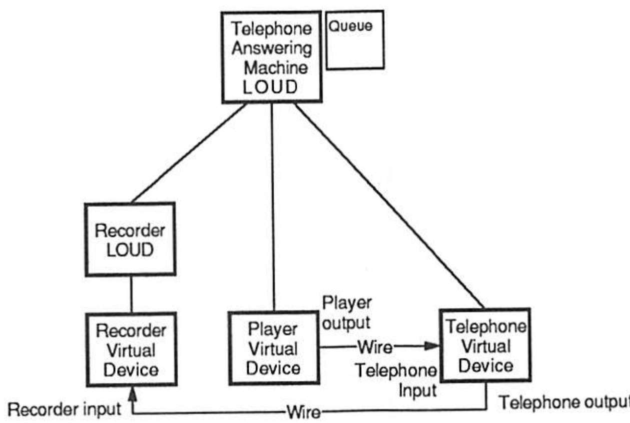


Figure 5-1: Answering Machine Loud

Virtual device classes. The devices supported by the protocol are divided into classes, which define generic audio functions that are supported by a set of device-independent *commands*. Below we have enumerated the classes supported by the protocol, and their commands. A device command is issued in either *queued* or *immediate* mode. Some device commands, such as `Play` or `Record`, must be synchronized with other commands, and can be issued only in queued mode. Other commands, such as `Stop` or `ChangeGain`, can be issued in either immediate or queued mode.[4] In immediate mode, a command takes effect instantaneously, and can stop processing of a queued command.

Inputs and *outputs* provide connections to external devices, such as speakers and microphones. The are used as wiring constructs to attach to the other classes. The base command is `ChangeGain`, which adjusts the volume.

Players have one or more output ports, typed according to a speech encoding format. They convert sound data to the output port type and then transmit the data out the port. The ports can be wired to an output device. The commands `Play`, `Stop`, `Pause`, and `Restart` control the transmission of the data on the ports.

[4]An application can issue a queued `ChangeGain`. An example would be a client that wants to play one sound after another but change the gain in between. In this case the client would issue a `Play`, a `ChangeGain`, and a `Play`, all in queued mode.

Recorders have one or more input ports, typed according to a speech encoding format. They store sound data received on the input ports. To digitize and store data from an external microphone, a recorder and an input of class microphone are wired together. The commands `Record`, `Stop`, `Pause`, and `Restart` control the flow of data from the ports.

Telephones are combined input and output devices with the commands `Dial`, `Answer`, `SendDTMF`, `Stop`, `Pause`, `Resume`.

Mixers take data on multiple inputs, combine the streams and then present the combined data on one or more output ports. The relative combination is determined by a percentage assigned to each input. The command `SetGain` sets the percentage to be mixed on a given input.

Speech synthesizers speak text strings. They have a single output for the synthesized audio. The commands `SetTextLanguage` and `SetValues` control interpretation of the text and acoustical characteristics of the vocal tract model used for synthesis. `SetExceptionList` allows applications to override the normal pronunciation of words, such as names or technical terms. `SpeakText` accepts commands to speak text strings.

Speech recognizers detect words spoken by a user. A recognizer has a single input, and produces recognition results as events. The commands `Train`, `SetVocabulary`, `AdjustContext`, and `SaveVocabulary` control which words a recognizer will detect, based on application and user.

Music Synthesizers process note-based audio. They accept commands, and produce audio data on their single output. The commands `SetState`, and `SetVoice` control music generation parameters. `Note` makes a sound.

A *Crossbar* is a switch to control routing of a number of inputs to a number of outputs. Each input can be connected to one or more of the outputs, as controlled by the command `SetState`.

A *Digital Signal Processor* is a set of software to manipulate one or more audio data streams. It may have several inputs and outputs. Commands have not yet been specified.

Device Attributes Both virtual and physical devices have *attributes* which describe specific features of the device. Attributes are used for virtual devices to constrain their mapping to physical devices, or to get information about the physical devices with which they are associated. Attributes of a physical device describe its actual capabilities.

To facilitate device-independence, an application specifies the desired virtual device by a list of attributes. The attributes can specify a device either tightly or loosely. For instance, a loose specification might be "give me a speaker". A more tightly specified list of attributes might be "give me the left speaker". The audio protocol maps the virtual devices created by the application onto the specific instances of those functional devices. When creating a virtual device, the application need only specify the class and other attributes of the device, rather than the specific hardware required to accomplish the operation.

The attributes for a specific device are dependent on the actual hardware. For example, the attributes of a recorder device include

1. sound encoding formats supported
2. whether the recorder supports automatic gain control (AGC) during recording
3. whether the recorder can compress the recorded audio by removing pauses
4. whether the recorder supports pause detection to terminate recording.

Attributes of a telephone device are largely constrained by the telephone equipment (digital or analog) and network (public or private) capabilities. Every telephone will have one or more numbers and area codes associated with it. Telephones may have multiple lines. Telephones may report information about incoming calls, such as the identity of the caller and whether the call was forwarded from another number.

An answering machine might wish to use calling party information to choose an outgoing message, or to label a message it takes. The answering machine client needs to query the device attributes of the telephone in order to determine whether this information will be available as part of the incoming call notification event.

What does the hardware do, really? Providing a device-independent interface is not specific enough for some applications and some hardware. Some devices are connected via physical wires that cannot be broken. Users will want global control over some devices, such as volume on a speaker, rather than the local control that virtual devices provide. A special LOUD tree, called the *device LOUD*, encapsulates all of the available functions in every device controlled by the server. The device LOUD tree contains a LOUD for every physical device, and if two devices are hard-wired, they are wired in the device LOUD. Each LOUD in the device LOUD is given a unique id that can be used by an application to monitor the device. The example in Section 5.9 monitors the telephone using the device LOUD.

Devices are controlled by applications through commands or through the manipulation of *device controls*. The commands provide a portable interface to abstract audio devices. Device controls allow for more complete access to devices at the cost of portability. Device controls have a format similar to X properties. Device controls should be necessary only for extensions, such as new subclasses of devices, or to take advantage of a particular implementation of a device at the cost of portability.

5.2. Wires

Wires establish the flow of data between virtual devices. They are used to construct complex devices by specifying connections between source and sink ports. A wire connects a source port of a virtual device to a sink port of another virtual device. This is how the internal connections of a complex device are established.

Wires have type information so that an application can query the type of the path, be it analog or a digital sample format. It is possible to query a virtual device for its wires, or a wire for its virtual devices and their corresponding port indexes. If it is necessary to constrain the nature of the path between two virtual devices, the desired wire type can be specified when the wire is created. The server checks that data on the wire matches the wire type.

In the device LOUD, the existence of a wire between two virtual devices indicates that there is a permanent connection between their respective devices. Unfortunately, not all hardware is as general as might be desired. If the capabilities and constraints specified for a virtual device require the devices to have permanently wired connections, special wiring rules apply. If an application attempts to attach a wire between a virtual device and any other virtual device, the capabilities and constraints for the connected virtual devices must match those of the devices that are physically connected, or an error will result. An example of such a mismatch might be an outboard speaker-phone that has hard-wired connections between a telephone line, a microphone, and a speaker. Attempts to wire a LOUD that requires use of one part of the speaker-phone with a device that cannot be implemented by another piece of the speaker phone is not allowed and will generate an error.

5.3. Mapping: associating a virtual device with an actual device

There is not necessarily a one-to-one correspondence between an actual piece of hardware and a virtual device. If a device can be used by more than one virtual device at the same time, the functional device will appear as multiple active virtual devices. For example, a speaker or output device, through which the sounds from multiple applications are simultaneously mixed, would be represented by multiple active virtual devices.

The server does not bind a virtual device to a physical device until the LOUD has been *mapped*. At this point, the server examines the attributes given when the LOUD was created to find a matching device. Most applications do not care which device they use, only that they can get the services they require. If an application must use a particular device, the id of the device in the device LOUD can be passed as an attribute to force the binding.

It is possible to augment the virtual device's attributes to tighten the device constraints. Such augmentation capability is useful when, for example, an application does not care which speaker it uses, but does not want to change speakers once the output has commenced. If an application requests a device by its class, the actual device used may change between activations. If it is desirable to use the same device for each request, an application can create a virtual device in a LOUD, and then map the LOUD. At this point, a `QueryVirtualDeviceAttributes` request will generate a list of device attributes that contains, among other things, the device ID selected by the server. This device ID can then be specified in an `AugmentVirtualDevice` request, so that it becomes an application-specified constraint.

5.4. Activation: who gets the device

LOUD access to shared resources is controlled by an *active stack*, which is the fundamental scheduling mechanism in the server. When a LOUD is mapped, it is put on the active stack. Unmapping a LOUD removes it from the active stack. The LOUD at the top of the active stack controls the functional devices represented by all of its virtual devices. Lower priority LOUDs can be put on the bottom of the stack to yield to higher priority LOUDs.

The server activates as many LOUDs as it can at one time. It does this by starting at the top of the active stack and activating all LOUDs that do not requires a resource that is being used exclusively by another active LOUD.

The activation and deactivation of a LOUD occurs dynamically. The state of the functional devices controlled by the LOUD are stored in its virtual devices, so that the server can restore the LOUD's devices to their state prior to the moment the LOUD was deactivated. Applications can request that a LOUD be activated or deactivated, and receive notification of these transitions. A mapped LOUD can be activated by the server or an audio manager at any time, so an application must treat a mapped LOUD as if it were active.

5.5. Synchronizing audio streams: command queues

Each root LOUD has a *command queue* to synchronize the actions of the virtual devices contained in the LOUD tree. Queues allow for the sequential processing of commands within the server, without requiring application notification and the associated round-trip communication. For example, an application may play a prompt and then record the user's response. In this situation, an application could first issue a `Play` to a queue and then issue a `Record`. The queue would process the `Play` and, on completion, start the record operation. Another case might be an application that wants to play several sounds back-to-back. The application could issue three `Play` commands, one right after another, and the server would play them in order, one after another, with the minimum possible space in between. For a set of digital sounds, there should be zero delay between them.

Queues have four possible states: *started*, *stopped*, *client-paused*, and *server-paused*. Time in a queue is relative to the activity of its LOUD tree. When a queue is paused, command queue relative time is suspended for that queue. If a LOUD is made inactive while processing a command, the server pauses the queue. Upon activation of a LOUD, a queue in the server-paused state is automatically resumed. Pausing a queue pauses the virtual device on which the current command is operating. All other devices in the LOUD are not affected.

The application can explicitly pause a queue by placing it in a client-paused state. The client-paused state allows a LOUD's queue to be paused, preempted, and re-activated without losing track of the queue state. The pausing and resuming of a queue are also propagated to all virtual devices affected by the current command. If the application issues a request to pause a queue in which the current command is operating on a device that cannot be paused, the queue is stopped.

There are four queue commands that allow device synchronization, but do nothing to devices. These commands are `CoBegin`, `CoEnd`, `Delay`, and `DelayEnd`. These queue commands are not meant to provide a programming language but to facilitate synchronization. There are no conditionals or branches and the queue is not an interpretor.

The `CoBegin` command causes all of the commands up to the bounding `CoEnd` command to be started simultaneously. The command after the `CoEnd` is not started until all commands within the `CoBegin/CoEnd` bracket are completed. These commands exist for the synchronization of complex audio configurations. A `CoBegin` command is useful when an application wants two operations to start at the same time. If, for example, an application is playing two sounds through a mixer, a `CoBegin` is necessary for the sounds to be started at the same time. The following example starts playing A and B at the same time. When both A and B are finished, sound C is started.

```
cobegin
    play A on device 1
    play B on device 2
coend
play C on device 1
```

The `Delay` command waits some interval time before processing. Commands contained within a delayed segment are processed sequentially, unless a `CoBegin` command is encountered before the `DelayEnd` command. The following example plays sound A, waits 5 seconds and then starts playing B. When B is finished, sound A is stopped.

```
cobegin
    play A on device 1
    delay 5 seconds
        play B on device 2
        stop device 1
    delayend
coend
```

5.6. Audio sound abstraction

Once a LOUD tree is created and the virtual devices are wired together, applications will want to pass audio data between the devices via wires. A *sound* is a typed object that represents digitized audio data. Its type is represented by the tuple (encoding, samplesize, samplerate). While the contents of a sound must be on the server side to be manipulated, the data can be supplied by the application or it may be supplied and controlled by the server.

The server provides a collection of sounds in its data space. Applications reference these sounds by name. The sounds are grouped into libraries or catalogues. Most sound data will be stored in files. However, some sound data will not be available to the server directly, but rather through an external device controlled by the server. Consider, for example, a CD that plays directly to its own speaker. In this situation, the sound data is supplied by the CD, rather than the application. Many CDs do not provide a digital data path to the computer so applications cannot read the audio data. Since the server controls the connection between the CD and the speaker, it must also control the sound data.

In addition, sound data can be supplied in real-time by an application, such as a networked-based audio process. The protocol provides a mechanism to supply or retrieve data from an active device. The application supplies the data to the server; it can then be used in the same way as server-side data.

5.7. Events

An *event* is data generated asynchronously by the audio server as a result of some device activity or as a side-effect of a protocol request. Events are the primary mechanism for synchronizing audio with other workstation services and media. The server generally sends an event to an application only if the application specifically asked to be informed of that event type.

There are 3 major event categories: *command queue*, *device* and *synchronization*. When a device or a command queue changes state, an event can be generated. For the queue, these are such things as `QueueStarted`, `QueueStopped`, and `CommandDone`. For devices, events are class specific. For the telephone class, they are ''a dial request has been issued'', ''the telephone has been answered'', ''the phone is ringing''. For the recorder class, they are ''start'' and ''stop''.

The synchronization events are used to coordinate the audio stream with other media or services. For example, consider an application displaying a set of images while playing a stored digital sound track. The images are displayed using the window system, and the audio track is played using the audio server. This application wants to display the images at some fixed rate. The application monitors the audio server synchronization events on the sound track, and uses them to time the update of the display.

5.8. Audio Manager support

The audio protocol provides several mechanisms for audio managers. It also specifies sensible defaults in the absence of an audio manager. These mechanisms directly parallel those provided by X. The mechanisms are: ambient domains, device exclusion, properties and redirection of mapping requests.

An *ambient domain* indicates a relationship between devices and the acoustic environment. A server supports at least one domain. For example, two speakers and one microphone on a user's desk are one ambient domain, called the *desktop domain*; sound from the speaker will be audible by the microphone. A telephone line is another ambient domain, as it does not interfere with the desktop domain. A speaker-phone is in both domains.

Ambient domains allow a user to activate a microphone, a device of class input, and exclude all devices of class output that might interfere with the same ambient environment. To accomplish this, in addition to the ambient domain attribute, devices of class input or output can request the attributes of *exclusive input* and *exclusive output*. Requesting a device with the exclusive input attribute preempts all other devices of class input in the same ambient domain. The exclusive output attribute performs similarly, but affects only devices of class output.

A *property* is a (name, value, type) triple. Properties can define any arbitrary information and can be associated with any LOUD or sound data. Properties can be used to communicate information between applications. In the case of an audio manager, they can be used to indicate application or user preferences. For instance, applications might attach a property named DOMAIN to the root LOUD. The value of this property would be the user's ambient domain preference for that application.

Another way to enforce policy is through *redirection control*. When an application attempts to map or restack a LOUD, the request may be redirected to a specified client rather than the operation actually being performed. In this way, the audio manager can override the application.

5.9. Example: an answering machine

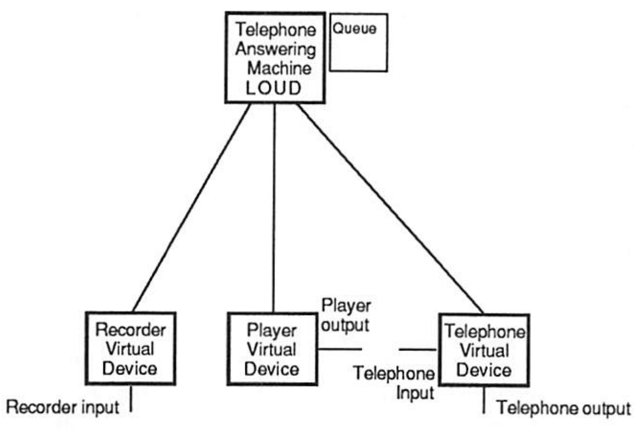


Figure 5-2: Answering Machine LOUD tree

To give a feel for how an application would use the protocol, this section contains an example of building and using an answering machine. An answering machine plays an outgoing or greeting message after answering a call, records an incoming message and then hangs up. The protocol entities needed to build the answering machine are a telephone, a player, a recorder and two sounds. The LOUD that the application would construct is shown in Figure 5-2.

When creating the virtual devices, the answering machine application specifies only the class of the device and the type of data that will be used. In this example, the greeting message is stored in an 8-bit µ-law encoding. Therefore, the attribute specification for the player is 8-bit µ-law. The application need not give any more information to the audio server. In this example, the incoming message is also an 8-bit µ-law sound.

The telephone will probably be an actual hardware device connected to the workstation. The player and recorder will be software devices, or algorithms. The player is responsible for reading cached sound data (probably from a file) and presenting that data on its output source. The recorder does the reverse; it takes data from its input sink and stores it as a sound (again, probably a file). At creation time, the three devices are unmapped and not activated. Any commands sent to them will be ignored until they are activated.

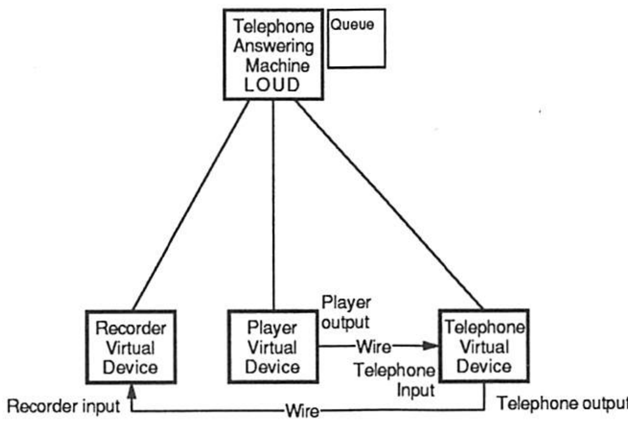


Figure 5-3: Answering machine: wired

The next step is to wire the devices to indicate the data paths. The results of wiring are shown in Figure 5-3. The output sink of the player is connected to the input of the telephone. This allows the greeting to be played to the caller. The output of the telephone is connected to the recorder's input source. This wire allows the caller's message to be recorded. As each wire is created and connected, the server checks the data types available at the two ends of the wire. If one end can only produce 8-bit µ-law and the other can only take ADPCM,[5] a protocol error will be generated.

Now that the LOUD is wired, it can be mapped and activated. Once mapped, the LOUD will accept and execute queue commands. To map and activate the LOUD, the audio server assigns virtual devices to actual devices, if possible. The assignment is qualified with an "if possible" because a device that is needed might be in exclusive use by another application. Or the LOUD may contain an impossible configuration, such as two virtual devices that specify the same actual device. In the later case, the map request will generate an error. If successful, the application gets an `ActivateNotify` event.

At last the LOUD is ready to accept commands. Commands are given to the LOUD, rather than the virtual devices (or, if there were any, sub-LOUDS) so that operations can be coordinated by the command queue. Playing the greeting before the phone is off-hook will result in a confused caller. The command queue for the answering machine is shown in Figure 5-4. First the phone is answered. When that command is completed, the greeting is played, followed by playing a "beep" sound. Once the beep is completed, the message recording begins.

Since most of the time the phone is not ringing, the LOUD can stay unmapped. The queue commands can be preloaded before the phone is answered. When the phone rings, the application would raise the LOUD to the top of the active stack, map it and start the queue.[6] The `Record` command has a termination

[5]Adaptive Delta Pulse Code Modulation, a compression algorithm, can reduce audio data rates by about one half.

[6]Because the answering machine LOUD is unmapped, the application cannot tell, from the LOUD, if the telephone rings. Therefore it monitors the device LOUD telephone.

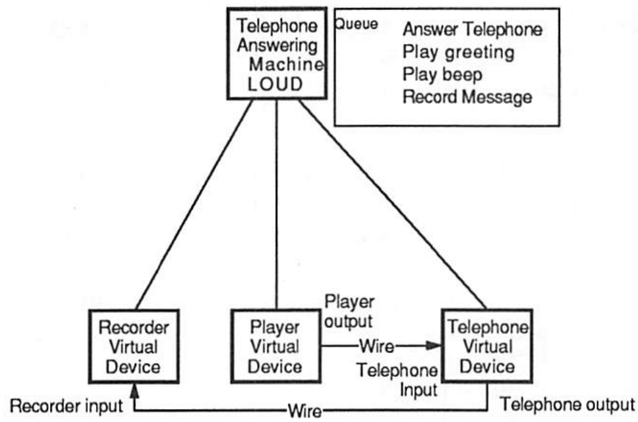


Figure 5-4: Answering Machine with command queue

condition, which can be either after a pause or when the caller hangs up.

Of course there are exceptions that must be handled. For instance, what happens when the caller hangs up? The caller may hang up before the beep is played. The LOUD can ask for `TelephoneNotify` events from the server. The application will get a `CallProgress` event that says that the phone is now hung up, and can then stop the queue and get ready for the next call.

6. A Prototype Implementation

A prototype audio server has been written. At the moment it supports playing, recording, and mixing, with the underlying LOUD creation, wiring, and mapping as described in the previous section. Multiple clients can play to the speaker simultaneously. The prototype server is written in C++ and is multi-threaded; it runs on a DECstation 5000, a 20 MIPS workstation, under UNIX. The audio hardware required is a simple CODEC with memory-mapped buffers.

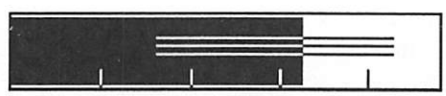


Figure 6-1: The SoundViewer widget supports audio playback and recording using several display modes

To test synchronization with other media, we have implemented a graphical sound viewer widget using the X toolkit and Alib. The widget displays a continually updated bar graph as a sound is played. Audio server synchronization events are used to control the graphics; the bar chart is updated in response to the these events. Figure 6-1 shows such a bar chart. The darkened area is the part of the sound that has already been played. The tick marks give an indication of the sound length. The dashes in the middle denote a part of the sound that has been selected, to be pasted into another application.

A working audio server and a few test applications have increased our confidence in the protocol and demonstrated the feasibility of a software architecture to support the server constructs described in the previous section. Although it would be premature to quote hard performance statistics, day-to-day use of the server indicates that performance will be well within our goals. We would like to be able to start playback of a sound, using an existing server connection, in less than several hundred milliseconds and support continuous playback without gaps, using well under 10% of the CPU.

But such numbers do not tell much about how well the audio medium will integrate with existing and future workstation application environments. With increased processor speeds and peripheral bandwidths, higher quality audio will become increasingly practical, but the limiting factor will be how well audio-based applications will perform in the context of many user applications running concurrently on top of a window system. Users are unlikely to tolerate system performance degradation for the sake of audio, as

they have yet to be convinced that audio has real utility. But until audio becomes universal, software vendors will have little motivation to write applications.

In order to meet what will be demanding requirements on generic multi-application support of audio, several aspects of our server have received particular attention. One is the use of a multi-threaded approach for modularity of design and management of multiple simultaneous audio data streams. The other is a set of design considerations to support the real-time nature of audio; some of these are exposed in the protocol while others are embedded in the server. These will be described in the remainder of this section.

6.1. Threads

We used threads in our implementation to allow concurrent use of the various audio devices. In addition, we wanted to be able to start and stop devices out-of-band. This section describes the various threads and what they do.

The *connection manager* detects and manages incoming connections. It is a daemon at a well-known port that detects incoming client connection requests and creates new connections for the clients. The connection manager runs as a thread. It is also responsible for initiating shut-down for a connection in case of errors. The connection manager keeps a container object for each client connection. The container objects hold everything that is related to a particular client connection.

The *device layer* contains devices that talk directly to the physical hardware. Each device runs as a separate thread. Devices provide command methods for others to call. Commands are handled synchronously. Events or status are reported by each device synchronously to its virtual devices.

The *virtual device layer* implements the virtual devices, each of which runs as a thread. Each virtual device, when active, is associated with an physical device. The different classes of virtual devices are subclasses of a common virtual device object class. Virtual devices process requests from the client input handler and associated command queue. Multiple virtual devices share a single device when possible; the server creates mixer-like virtual devices when it can, transparent to applications.

Data source and *data sink* implement the server version of wire, manipulating data inside the server. They create an efficient data path by by-passing in-between devices and connecting the source and sink objects directly. Each source and each sink runs as a separate thread.

6.2. Copying and Caching Data

Implementing an audio server on a non-real time, general-purpose UNIX system poses interesting challenges. The biggest and most obvious problem is the real-time nature of audio. Although medium quality audio data rates are not excessive, substaining the rate may be difficult. Sub-second delays in the middle of speech are annoying and damage the intelligibility of the speech. Another challenge is supporting seamless transitions between commands in the queue, so that playback appears continuous.

Our design alleviates some of the real-time issues and allows the client to deal with those cases where the system falls short. Previous experience with X has shown that we can often achieve real-time appearance with sufficient buffering and an intelligently designed protocol.[7] Telephone or voice quality audio requires a 8000 bytes per second data stream. Higher quality audio can involve data rates up to 175,000 bytes per second. The lower data rates are usually adequate for telephone quality speech and within the ability of current technology, so this is our focus. As coding standards emerge, higher bandwidth voice coding with only modest increases in data rate will become more common and will be supported by the server; these data rates are already supported by the protocol.

We have addressed the real-time constraints in several ways. The most common operation is the playback of recorded audio data. If the data is cached by the server, for instance in the file system, the data transfer is local, the number of copies small, and the performance should be acceptable. If the application

[7]It was originally believed by many that rubber-banding and menu tracking require extensive system changes. In the early days of X it was shown that this is not the case.

wants to supply real-time data to the server, the constraints are harder to satisfy. When an application is providing data in real-time there is the possibility that the application or the application's source, maybe a network connection, will not have the data when it is needed. To deal with this, the protocol provides client-side reading and writing of data. This type of interface allows an application to implement its own policy and has been proven effective in the past [1]. This approach also allows the application to make trade-offs with respect to latency and memory.

One requirement of the protocol is to support seamless transitions between commands in a queue. There are two interesting cases to consider: playing back to back sounds, and a play followed by a record. These can be illustrated by considering the answering machine example in Section 5.9. When the phone rings, the application starts the queue and the server executes the queued commands. Once the outgoing message play is started, arrangements must be made so that the beep sound starts instantaneously after the message.

Once the queue starts, the server answers the phone and starts the play. When the first play command is about to finish, the player device informs the queue of the time at which the last sample will be played. The queue can then issue the next play command specifying that the play should start when the first command is scheduled to terminate.[8] Pre-issuing commands allows plays to occur without a single dropped or inserted sample. Recording back-to-back with a play is accomplished in the same manner.

7. Conclusion

This paper has discussed a server to control audio and telephony hardware. We described the requirements of an audio server, a protocol to communicate with the server, and a set of constructs which an application must manipulate to use the server. We described an implementation of a prototype server, with emphasis on its multi-threaded architecture and design goals for real time performance.

This work is in its early stages, but we are encouraged by our progress so far, and are confident of the utility of the server model to support time dependent media.

8. Acknowledgements

Our protocol owes many ideas to pioneering work done on the VOX project at Olivetti Research Labs. We would also like to thank Chris Kent, Tom Levergood, Larry Stewart and Jim Gettys for their comments and review of our protocol. Mark Ackerman of M.I.T. wrote the original graphical SoundViewer widget software. Chris Kent and Gordon Furbush provided valuable comments on drafts of this paper.

References

1. P. Zellweger, D. Terry and D. Swinehart. "An overview of the Etherphone system and its applications." *Proceedings of the 2nd IEEE Conference on Computer Workstations* (March 1988).

2. C. Schmandt and M.A. McKenna. "An audio and telephone server for multi-media workstations." *IEEE Proceedings of the 2nd Workstations Conference* (March 1988), 150-159.

3. Robert W. Scheifler, James Gettys, and Ron Newman. *X Window System.* Digital Press, Bedford, MA, 1988.

4. Barry Arons, Carl Binding, Keith Lantz, and Chris Schmandt. "A Voice and Audio Server for Multimedia Workstations." *Proceedings of Speech Tech '89* (May 1989).

---

[8]Note the queue does not calculate the ending times itself, because the server's CPU may not use the same time base as the CODEC's, and clock skew is a problem.

Susan Angebranndt received a B.S. in Mathematics from Carnegie Mellon University in 1980. She is currently a Consulting Engineer at Digital Equipment Corp. and technical project leader for the Multimedia software development group.

Richard L. Hyde received a B.S. in Computer Science from Brigham Young University in 1982 and an M.S. in Computer Science from Brigham Young University in 1983. In 1984 he join Digital's UNIX engineering group. In 1987 he moved to Digital's Western Software Labs to help with the X11 development effort. He currently is the technical project leader for the audio server work in the Multimedia software development group.

Daphne Huetu Luong received her degree in Electrical Engineering and Computer Science from the University of California, Berkeley in 1987. She is a software engineer at Digital Equipment Corporation's Western Software Laboratory in Palo Alto, California. She works in the Multimedia software development group. She previously worked at Xerox, doing network communication software development.

Chris Schmandt received his B.S. in Computer Science from MIT and an M.S. in computer graphics from M.I.T.'s Architecture Machine Group. He is currently a Principal Research Scientist and director of the Speech Research Group of the Media Laboratory at M.I.T.

Nagendra Siravara recieved an M.E. in Electrical Engineering from Indian Institute of Science, Bangalore, India. Since 1990, he has worked in the Multimedia software development group at Digital Equipment Corp. He worked previously on software for voice response applications, speech and image processing and error correction coding.

A Brief Overview of the DCS Distributed Conferencing System

R. E. Newman-Wolfe, C. L. Ramirez, H. Pelimuhandiram,
M. Montes, M. Webb and D. L. Wilson
Computer and Information Sciences
CSE-301
University of Florida
Gainesville, FL 32611
nemo@cis.ufl.edu

Abstract

The Distributed Conferencing System (DCS) at the University of Florida is a distributed package providing real-time support for cooperative work.[1] In this system, a set of mechanisms for conference management supports a wide range of floor control paradigms. The small, flexible message-passing interface to the conferencing management processes permits shared applications to be installed easily in DCS. Currently, DCS has applications that support concurrent development of text and graphic documents; remote demonstration, testing and debugging of programs; and automatic creation of transcripts of meetings including motions made and voting results.

1. Introduction

Research in distributed conferencing and shared applications has usually focused on a particular application, or on sharing an existing application without modifying the application code. We believe that cooperative work generally involves several kinds of shared applications, and that the framework for coordination of effort is significant in itself. Sharing should not be transparent for all applications, nor is the same architecture appropriate for all applications. DCS not only provides a cohesive set of shared applications suitable for collaborative preparation and demonstration of text, figures and software, but also a rich set of mechanisms for directing the collaborative effort. DCS monitors user behaviors to assist our study of how people interact when using these distributed collaboration tools.

DCS is written using Unix[2] 4.3bsd sockets [Bach 86] [Leffler et al. 89] and XWindows [Johnson & Reichard 89] with the X Athena Widgets package. It runs concurrently with other tools; thus other programs and sources of information are available to conference participants. Versions of DCS have been running at the University of Florida since August 1990.

A unique aspect of DCS is the voting mechanism, which provides flexible, decentralized control of conferences. Within the text editing and graphics editing applications, fine granularity write and view

[1] This work is partially supported by the University of Florida - Purdue University Software Engineering Research Center.

[2] Unix is a trademark of AT&T, of course.

locks are provided (see below). Each user may define the degree to which she wishes to share views, or see ongoing work. This is a generalization of What-You-See-Is-What-I-See (WYSIWIS) [Stefik et al. 86, 87] that we call "What-You-See-Is-What-I-May-See" (WYSIWIMS). The WYSIWIMS paradigm is similar to the filtering provided in Suite, which propagates changes depending on syntactic or semantic integrity [Dewan & Choudhary 90]. Discussion windows facilitate communication among members of a conference by allowing a channel of "out-of-band" communications. A transcript of the discussion is preserved, allowing both asynchronous and synchronous cooperation. Users have access to control functions and information through a status window. In addition to these control features, DCS provides a suite of applications with the flexibility to add others easily.

The remaining sections briefly describe related work in groupware [Ellis & Gibbs 88], and some details of the structure of DCS. The abstract is necessarily brief, and more thorough explanations may be found in [Ramirez 91], [Wilson 91], and [Newman-Wolfe et al. 91].

2. Related Work in Groupware

Earlier systems for distributed collaboration date back to the 1960's and Engelbart's work with NLS [Engelbart & English 68] and have continued through MCC's Project Nick [Begeman et al 86]. Most office automation systems focus on only one application, and those that belong to a package usually have limited facilities for supporting shared work. Typical applications include: shared calendars and schedulers (RTCAL, MTCAL [Greif & Sarin 86]), shared draw screens (NLS [Engelbart & English 68], MBlink [Sarin & Greif 85], Colab [Foster & Stefik 86, Stefik et al. 87], Commune [Bly & Minneman 90], TWS [Ishii 90]), shared graphics editors (Xsketch [Lee 90]), shared text editors (Collaborative Editing System (CES) [Greif & Sarin 86, Sarin & Greif 85], Shared Books [Lewis & Hodges 88]) and hypertext browsers (Contexts [Delisle & Schwartz 86], NoteCards [Trigg et al. 86], TEXTNET [Trigg & Weiser 86], Intermedia [Garret et al. 86]). Some of these offer more than one medium for communication and cooperation. Another approach uses shared terminal emulators and other facilities for sharing existing serial applications without modification (Cantata [Chang 87], Augment [Engelbart 84], Dialogo [Lantz 86], Rapport [Ahuja et al. 88a], Timbuktu [Farallon 88]). These independent shared applications are more tightly coupled in DCS, reflecting the cohesive nature of the conference. DCS more closely resembles Share [Greenberg 90], MMConf [Crowley et al. 90], and Mermaid [Watabe et al. 90] in dealing with conferences as a whole. An excellent introduction to work in this area may be found in [Greif 88]. Additional sources include the proceedings of the conferences on Computer Support for Cooperative Work (CSCW 86, 88 and 90) as well as the Conference on Organizational Computing Systems (nee Conference on Office Information Systems).

3. DCS Description

3.1. Goals of DCS

DCS provides a distributed, real-time conferencing facility for the cooperative creation, review and execution of papers, proposals, and programs. Conferences may be long-term or short-term and are dynamic in that their constituency may change over time; they may also split or merge. Conference control is hierarchically democratic, and DCS's architecture is hierarchically distributed.

DCS uses Unix 4.3 sockets [Bach 86], [Leffler et al. 89], [Kernighan & Pike 84] and X-Windows [Johnson & Reichard 89] for implementation to allow it some degree of independence. By using "middleware" DCS will be able to run on a heterogeneous, distributed environment. Porting the system should be easy.

Shared applications may be made accessible within DCS by adherence to a small, message-passing interface within the application code, and by providing a minimal amount of information to DCS regarding the application name and usage. This allows DCS to be extended and customized.

Our intent is not only to provide an easily portable and extensible conferencing system, but to study the ways in which people interact when using a distributed conferencing facility of this type. From this, we hope to learn what features are critical to the success of groupware, and what resources are needed to support them. To this end, we are currently incorporating monitoring code in DCS to measure system performance parameters (such as number of messages, message size, response times, etc.), user behaviors (facilities used, conference interactions, preferences) and conference behaviors (control modes used, time spent sharing objects, lock collisions, lock preemptions). Direct user feedback will also contribute to our evaluation of DCS.

3.2. Functional Description of DCS

Important concepts in DCS are conference typing, user roles, voting, and independence. Conference types define the desired level of stability of the conference and how long the conference will last. User roles describe what a conference member may do. Voting resolves conference control and user defined issues. Finally, independence means that each user has control over the degree to which she shares with others.

DCS provides two types of conferences: short-term and long-term. A long-term conference persists, even when members are not logged in, until it is terminated by the last member resigning. Over this period, a record is kept of the members, the files, and access information. An effort is made to keep conference objects stable over system disruptions. Short-term conferences have no such records; they are automatically cancelled when the last member exits.

Within a conference, members may have the role of a voter, a non-voter or an observer. The roles are created to maximize the number of participants, yet preserve the integrity of the conference by limiting rights given to members. Voters will be permitted to participate in group decisions about conference actions, such as whether to allow a new user to join the conference, or what the disposition of a conference object should be. Both voters and non-voters have read and write access to all files within their conference. Observers are denied write access, but may view conference files and other objects.

All conference control activities are resolved using a general voting mechanism. These actions include changing the role of a user, merging two conferences, splitting a conference, and disposing of conference objects. In addition, DCS supports user-defined motions, which are not interpreted by the system but are only recorded and handled through the voting mechanism.

Off-line voting is the general control paradigm, so that the conference need not be suspended while a vote is under way. When a motion is made, a small dialog box with the motion and a checklist appears on

the screen of each active user. The user may then vote for or against, abstain, or defer on the motion. Unresolved motions are placed on a referendum list and members may vote on any motion for which they have not yet voted (deferral is not a vote). The referendum list may be examined by any member at any time. Once enough votes for a motion have accumulated, action may be taken. Conversely, if enough negative votes accumulate against the motion, then it will be removed from the referendum list and no action will be taken. Voting may be rollcall or anonymous, short circuit or exhaustive. The results of voting are appended to the conference control log. By adjusting the roles of users, conference control ranges from dictatorship (one voter) to total democracy (all members are voters) to anarchy (no voters).

3.2.1. Extra-conference Activities

DCS provides some operations to users who are not active participants in a conference. Any user may ask for a list of conferences, initiate a conference, or request to join a conference. All operations other than direct conference file access through usual Unix mechanisms require the user to invoke DCS.

Each conference has its own directory with its own files. Members of the conference may access these files through DCS without having access through Unix [Kernighan & Pike 84]. Conversely, any user may gain access to conference files according to the usual Unix access modes without involving DCS. The default mode is no access, but the protection mode may be changed by the members of a conference.

3.2.2. Intra-Conference Control Activities

Within a conference, a member may make queries, make motions, and vote. Queries allow the member to review the status of the current conference or other conferences without leaving the current conference. Motions are either requests for actions within a conference, or are user-defined and are not interpreted by the system. Motions are usually subject to voting by the members (see above).

Control activities within a conference consist of changing user roles (joining, resigning, upgrading to voter, etc.); sending invitations to join; splitting and merging conferences; changing conference parameters; and determining the disposition of conference objects.

3.3. DCS Architecture

DCS is implemented as a group of cooperating processes. We term these the Central Conference Server (CCS), the Conference Manager (CM), the User Manager (UM). Of these, only the CCS need always exist and only the CCS needs to have an easily found port. Each active conference has a CM, and each user active in a conference has a UM. In addition, each application has its own logical Application Manager (AM) and architecture internal to itself. Currently, the applications we have developed all have an AM process with an application user process for each user working with that application, but this architecture is not required.

3.3.1. The Central Conference Server

The Central Conference Server (CCS) is a daemon that listens at a fixed address known to the process invoked by users. It provides coordination between and access to conferences. The CCS also monitors the performance of CMs and does error recovery.

When a user first invokes DCS, she obtains a UM that runs locally and is connected to the CCS. Only the status window is displayed, and the CCS may be queried about existing conferences or the user may request to join or create a conference. Requests to join a conference are forwarded to the appropriate CM, which will allow members to connect immediately, or will make a motion to allow a new user to become a member. Create requests cause DCS to prompt the user for descriptive information (for external listing) and conference type. The CCS then creates a new CM and a new directory, and connects the user's UM to the new CM.

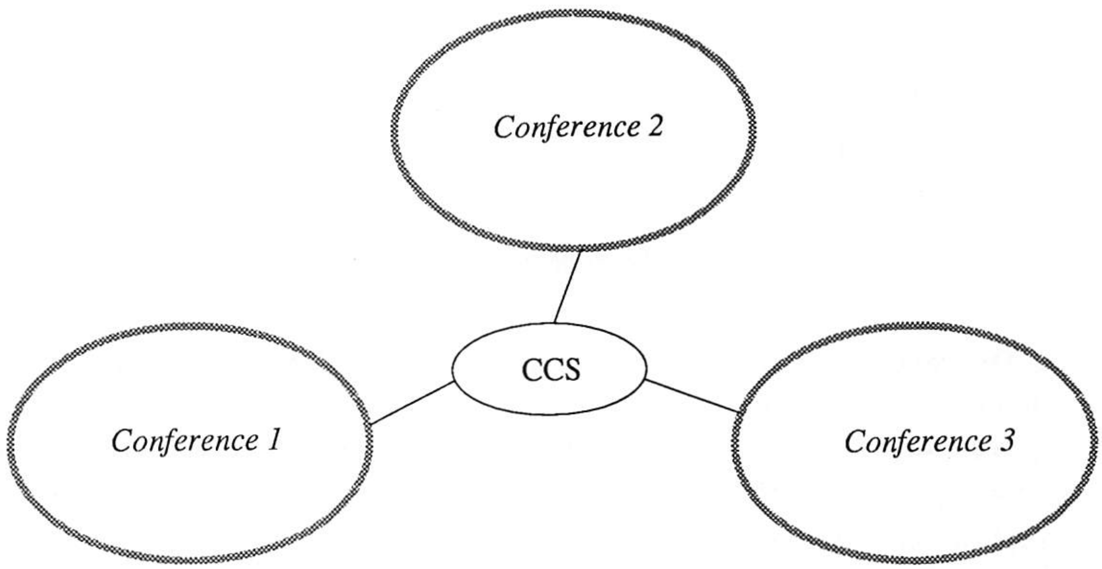


Figure 1. CCS connections to multiple conferences.

3.3.2. The Conference Manager

Each active conference has a Conference Manager (CM). The CM is responsible for providing basic conference control and for the discussion window. The main access point to the CM is the status window. Through the status window, the user may perform conference control activities described earlier, make queries, or request application windows. When an application window is requested by a user, the CM must determine whether an appropriate AM exists, invoke one if it does not exist, and connect the user to that AM. The CM also provides the AMs with access to voting and other conference control mechanisms through the message-passing interface. Finally, the CM is responsible for checking on AMs and the CCS to verify their operation.

3.3.3. The User Manager

The User Manager (UM) acts as a dialog manager for the individual user active in a conference. It routes information between the user's input devices, the CM, and the user's output devices. When the user requests an application window, the UM asks for information needed by the CM to connect the user to the

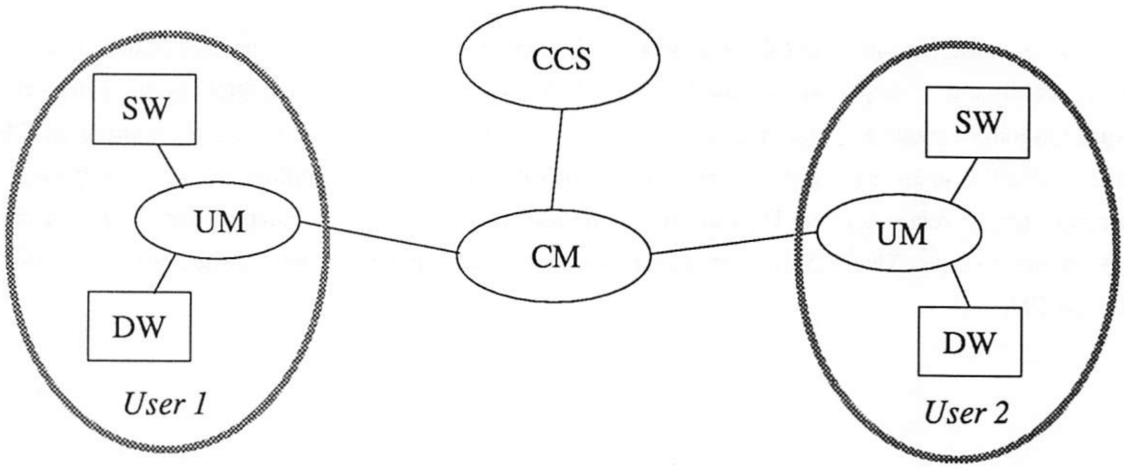


Figure 2. CM connections within a conference.

right AM. Once the CM insures the existence of the appropriate AM, that AM's port is sent to the UM along with instructions for invoking the application window (AW) that will be connected to the AM. Once the UM starts the AW, it is no longer concerned with that application, leaving all the intra-application communication to be handled by the AM and AWs attached to it.

3.3.4. Application Managers

Each application in DCS has its own (logical) application manager. There may be only one AM for a particular application in a conference, or there may be several for the same application within a conference. For example, the execute window has only one manager per conference regardless of the number of execute sessions in progress. On the other hand, the graphics edit application has one Graphics Manager (GM) for each graphics file that is being edited, so there may be several GMs for a single conference. Conceptually, each graphics file has a GM and together, these form a graphics object.

By having AMs, the design of DCS is modular: stand-alone shared applications are easily introduced into its framework. This also distributes the workload within DCS in two ways. First, each process that handles multiplexing within an application can be relatively simple since they only handle a fraction of the communication load. Second, a new AM may be started on a different machine from the CM or existing AMs, so no one machine will become overloaded.

While the shared applications we have developed so far have all had an explicit AM connected to AWs running on user's workstations, this architecture is not required by DCS. DCS does require that each application have a liaison communicating with the CM that acts as a virtual AM. This is needed for instance to clean up orphaned AWs when a user quits DCS without closing AWs first. Since the application may not have a fixed AM, but must have some internal communication abilities, DCS provides a message type for migration of the logical AM if needed. Thus an application could be a set of peer processes that pass AM duties among themselves as the set changes.

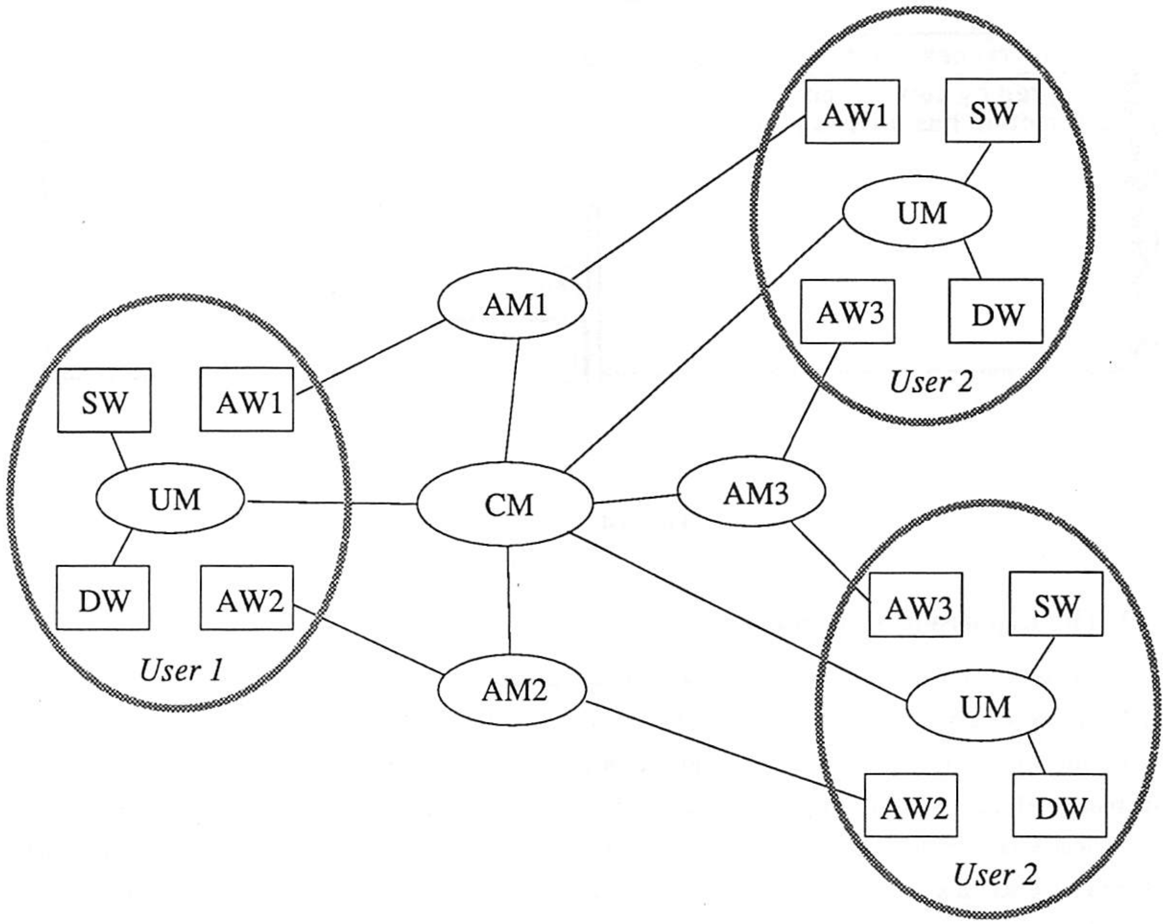


Figure 3. Application managers and application windows within a conference.

3.4. Base Windows

A user running DCS always has a status window. If the user has joined a conference, then she also has a discussion window. These are the base windows of DCS, and they provide support for distributed decision making and conference control.

3.4.1. The Status Window

The status window provides conference control and information. It is the only window that always exists, whether or not a user is inside a conference. Members may make motions, request information, and vote through the status window and its menus. Notices of actions that are requested, pending, or completed, as well as requested information, are displayed in the status window. The output of the status window that is relevant to the whole conference is buffered in a file as a record of conference actions. Users may scroll within the status window independently of one another.

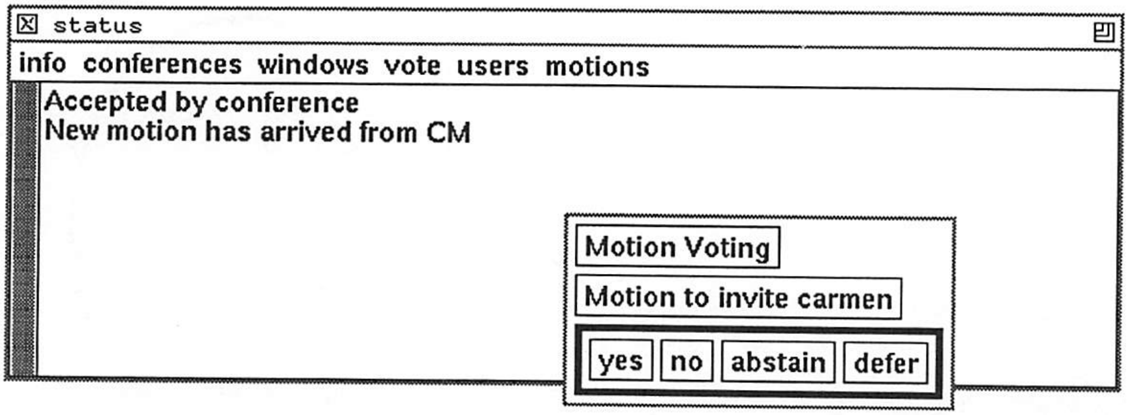


Figure 4. Status window.

3.4.2. The Discussion Window

Discussion windows allow real-time communication among collaborators. Each member has access to a discussion window that is local to the conference to which she belongs. The window, which is similar to a *gossip* window [Ramirez & Pelimuhandiram 90] is opened by default when the member joins the conference. All members of a conference have append-only rights to the top half, which may be viewed synchronously or asynchronously. By default, the user's login name is prepended to all contributions she makes to the discussion window, but an anonymous mode is available to omit this if attribution would inhibit free discussion.

3.5. Application Windows

In addition to the base windows, several application windows may be brought up in DCS from a menu on the status window.

3.5.1. The Execute Window

Every member of a conference may create an execute window through which she has access to a shell. Currently, users may only execute a program producing line-oriented ASCII output and taking serial ASCII input. This window may be exported to other users, allowing them to see the execution of the program taking place in the source execution window. The execution window is the only window with a conference pointer. Only one member at a time controls the pointer and the input of each execution window. This seems to be a necessity for maintenance of consistency when serial applications are shared.

When the execution window is opened, the input control to the window belongs to the user who opened it. The user who has control of the input may pass control to another user at any time. The controlling member may specify which other active members may view the execution window, except that the member who created the execution window cannot be excluded. All other windows are synchronized in a WYSIWIS fashion to reflect every input and output of the controlling window.

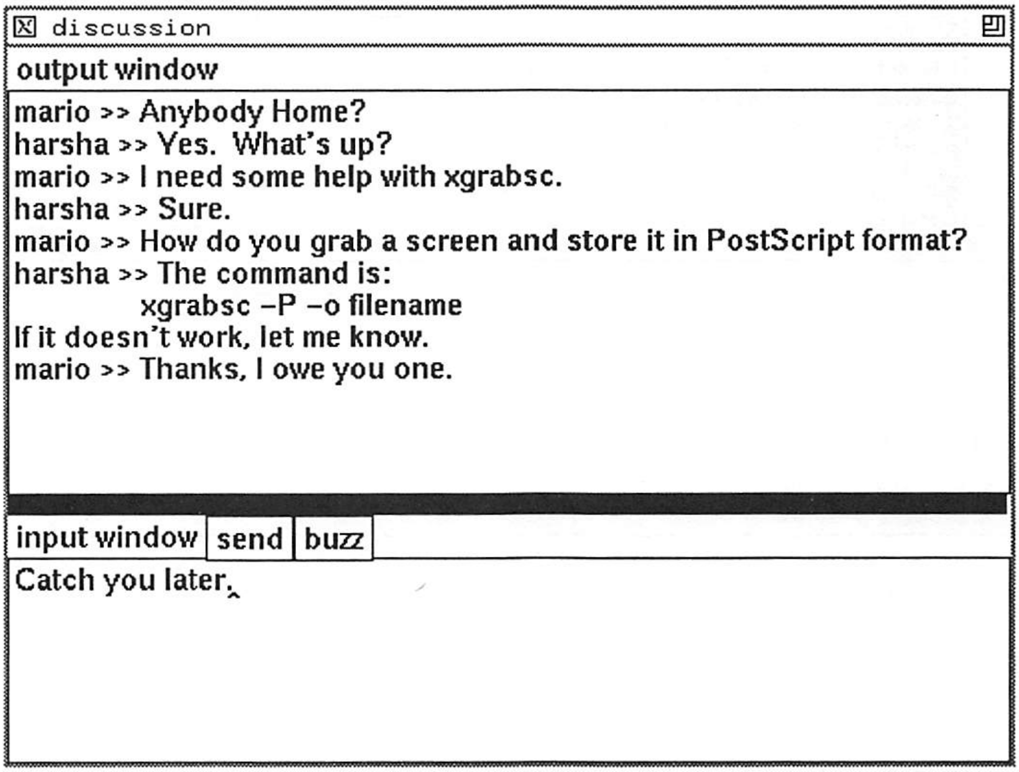


Figure 5. Discussion window.

Since only the output of the program's execution is sent to the slave execution windows, people can participate in conferences in spite of some differences in software and hardware [Ahuja et al. 88a],[Ahuja et al 88b]. This tool provides the facilities for activities such as joint debugging of programs, and conducting tutorials or demonstrations remotely.

3.5.2. The Text Edit Window

The DCS system provides a concurrent text editor, permitting concurrent access to a shared document using fine granularity locks. Locked portions of text may begin and end between any pair of adjacent characters in the file, or at its beginning or end.

Two types of lock are used: the write lock and the view lock. The write lock is allocated on a first-come, first-served basis and it ensures exclusive write permission to a section of the file. Only one write lock may be held per text edit window. A write lock is held by the writer until it is explicitly released, at which time the changes will overwrite the old version of the section. The user holding a write lock may also place a view lock on the locked section. View locks prevent other users from 'looking over the shoulder' of the writer while the edit is in progress (i.e., neither update nor synchronous viewing of the

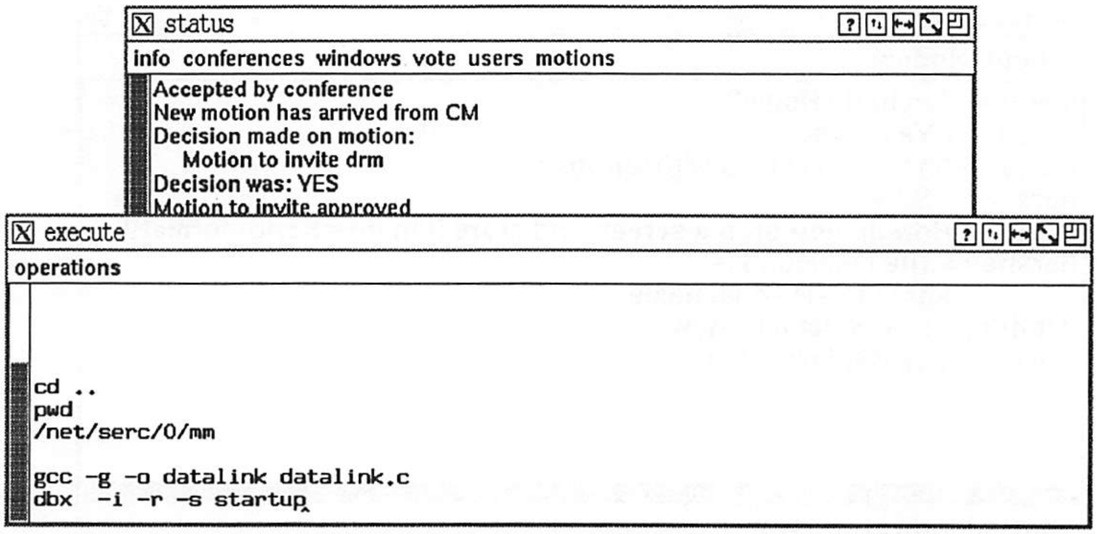


Figure 6. Execute window.

edited section is allowed). They do not prevent other users from viewing the 'old' version of the text in the locked region.

User have three modes available for viewing the contents of a text file: static, snapshot, and synchronous. Static viewing allows locked portions of files to be viewed as they were before modification began. Snapshot viewing causes the member's window to capture a snapshot of the current state of a locked portion of a file, while synchronous viewing causes changes to be seen as they occur. If the reader is viewing synchronously, she is not able to scroll within or edit the file until she returns to an asynchronous viewing mode. These latter two modes are disabled if the source is view-locked. By default, the reader views the text in static mode, seeing updates only when locked portions of text are released.

3.5.3. The Graphics Edit Window

Users may invoke a concurrent, object-oriented graphics editor from DCS. The stand-alone version of this editor is called Ensemble, and it is based on the tgif public domain graphics editor from UCLA. Objects may be manipulated in the usual ways.

Locking is used to prevent conflicts in the graphics editor. Locks are placed implicitly when a user selects an object: if a locked object is selected, an error message is printed in the banner and the selection is rejected. Changes are shown when the selection (and thus the lock) is released.

Ensemble allows users to preempt locks held by others, in the event that a user holds locks on some objects for too long while another user wishes to work with them. While we initially handled lock preemption through voting, it was cumbersome to obtain sufficient votes quickly enough to be worthwhile.

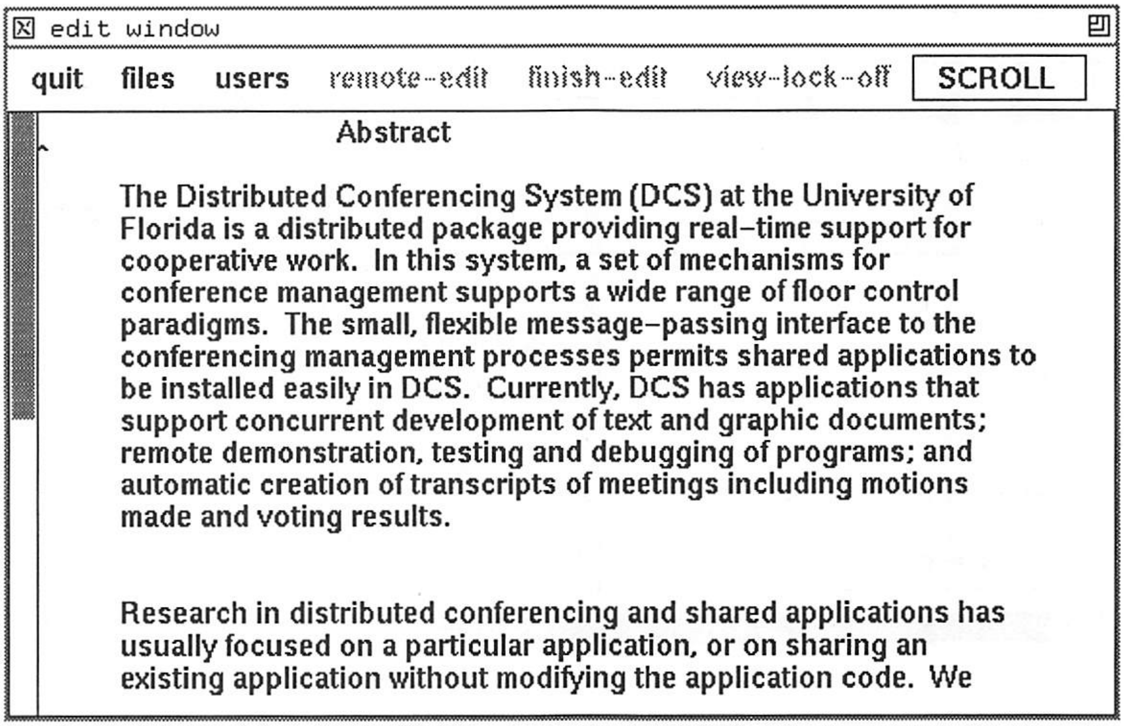


Figure 7. Text edit window.

Instead, any editor in DCS may preempt locks at will, and we rely on their desire to cooperate to damp these urges.

Each user has a pointer, which may be exported to other users. Each user also has the option of viewing other users' exported pointers. A user importing pointers will see another user's pointer only if the other user exports her pointer. Pointers may be used for gesturing while discussing the contents of a graphics window, a process that we found demanded an audio connection as we note in the conclusions section. Although we attempted to avoid use of special equipment, close interaction while editing a canvas is one instance in which it is very desirable.

When a graphics edit window (or Ensemble window) is opened within DCS, it is automatically and transparently connected to any other Ensemble windows editing the same file. These AWs coordinate through the Graphics Manager (GM), one of which exists per graphics file for which an Ensemble window is open. The GM maintains the last stable version of the file, coordinates locking and updating, and multiplexes pointer information. Each Ensemble window process has its own copy of the object data base that it manipulates locally for display purposes, sending and receiving object updates when necessary by exchanging messages with the GM. Since messages are received serially by the GM, there are no ties and all requests are handled in a first-come, first-served manner. The rate of information exchange is very low compared to systems in which low level information is broadcast to application windows. Each AW may also deal with the updates in a manner appropriate to the section of the canvas displayed locally, since each Ensemble window is able to view the canvas independently. This provides excellent response time for

edits performed locally, with good response time for those performed remotely.

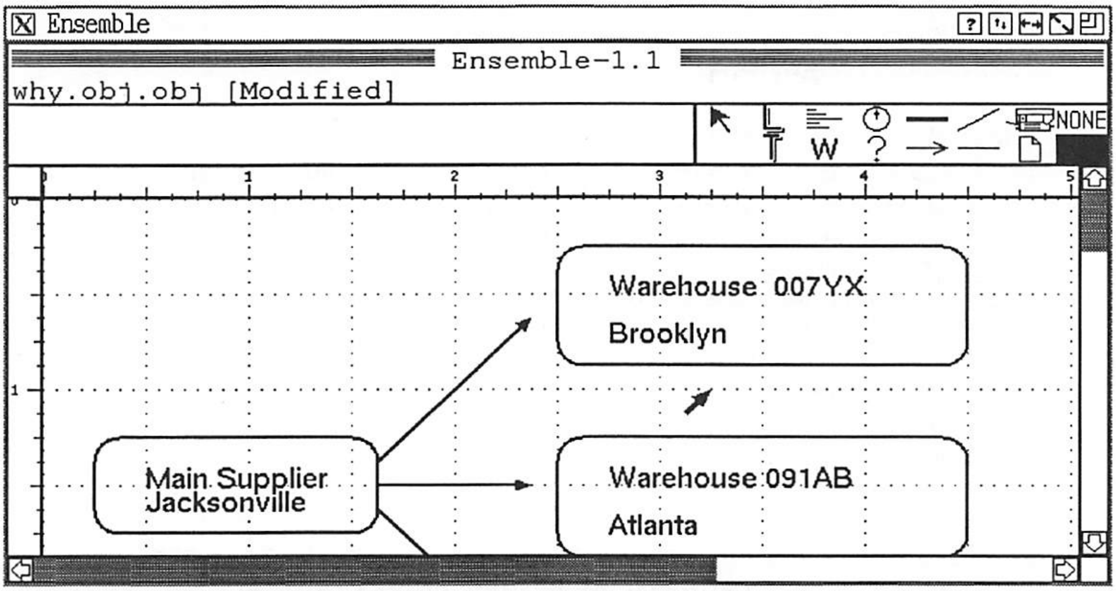


Figure 8. Graphics edit window.

4. Conclusions

DCS is a distributed real-time conferencing system running at the University of Florida. It provides flexible conference control and has the concept of multiple conferences. Within DCS, fine-grained concurrent text and graphics editors are available, as well as a discussion window and a general-purpose execute window for sharing a limited class of serial applications. Applications planned for DCS include software engineering tools (a code review tool) and a more general execute window.

Applications within DCS use a variety of paradigms for access: immediate and implicit access is granted for appending to the discussion and status windows, exclusive floor control is passed serially in the execution window, fine grained locks are used in the graphics and text edit windows. We have not found locks to be restrictive, as has been feared [Ellis & Gibbs 8]; this may be due to the selection of the applications in which locks are used and the capability to remove locks that have been held 'too long.'

Through the paradigm of the Application Manager, an interface to the Conference Manager of DCS, each shared application can implement the architecture most appropriate to its functioning. New applications are currently easy to add to DCS, provided that they follow an undemanding message passing protocol. We plan to automate the installation process so that even the small changes to the program required on addition of a new application may be performed quickly and correctly.

In using DCS we found occasions when an audio connection among users would have been desirable. For example, gesturing with pointers in a graphics window while conversing in the discussion window is clumsy. In future versions of DCS, we plan to incorporate a voice channel for communication that will enhance the interaction among users.

Our experience with DCS is still preliminary, and we intend to perform extensive monitoring in order to determine which features are most desirable and which are not. DCS gathers information on user behaviors and system performance with internal statistics gathering code. It is dangerous to presuppose the features or functionality that users will find most useful, so we plan to beta test DCS soon. The cost of using DCS will be to respond to a user evaluation questionnaire after three months of use. This information will guide future development of DCS.

5. References

[Ahuja et al. 88a]

Ahuja, S., Ensor R., and Horn D., "The Rapport Multimedia Conferencing System," Proc. of Conf. on Office Information Systems, Palo Alto, March 1988, pp. 1-8.

[Ahuja et al. 88b]

Ahuja, S., Ensor R., and Horn D., "Networking Requirements of The Rapport Multimedia Conferencing System," Proc. IEEE Infocom 88, New Orleans, March 1988.

[Bach 86]

Bach, M. The Design of the UNIX Operating System Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Begeman et al. 86]

Begeman, M., Cook, P., Ellis, C., Graf, M., Rein, G. and Smith, T., "Project Nick: Meeting Augmentation and Analysis," Proc. of Conf. on Computer Supported Cooperative Work, Austin, December 1986, pp. 1-6.

[Bly Bly, S. A. and S. L. Minneman, "Commune: A Shared Drawing Surface," Proceedings of ACM COIS 90, pp. 184-192.

[Crowley et al. 90]

Crowley, T., Milazzo, P., Baker, E., Forsdick, H., and Tomlinson, R., "MMConf: An Infrastructure for Building Shared Multimedia Applications," Proceedings of the Conference on Computer Supported Cooperative Work, Los Angeles, Oct. 1990, pp. 329-342.

[Delisle & Schwartz 86]

Delisle, N., and Schwartz, M., "Contexts-a partitioning concept for hypertext," Proc. of Conf. on Computer Supported Cooperative Work, Austin, December 1986, pp. 147-153.

[Dewan & Choudhary 90]

Dewan, Prasun and Rajiv Choudhary, "Flexible User Interface Coupling in Collaborative Systems," Purdue Univ.-Univ. of Florida Software Engineering Research Center, SERC-TR-76-P, June 1990.

[Ellis & Gibbs 88]

Ellis, C. A. and S. J. Gibbs, "Concurrency Control in Groupware Systems," MCC Technical Report STP-417-88, December 1988.

[Engelbart & English 68]

Engelbart, D. and W. English, "A Research Center for Augmenting Human Intellect," Proc. of FJCC 33(1):395-410, AFIPS Press, 1968.

[Engelbart 84]

Engelbart, D., "Authorship Provisions in Augment," IEEE Compcon Conference, 1984.

[Farallon 88]

Farallon, Timbuktu User's Guide, Farallon Computing, Inc., Berkeley, CA, 1988.

[Foster & Stefik 86]

Foster, G. and Stefik, M., "Cognoter, Theory and Practice of a Colab-orative Tool," Proc. of Conf. on Computer Supported Cooperative Work, Austin, December 1986, pp.7-15.

[Garret et al. 86]

Garret, N., Smith, K., and Meyrowitz, N., "Intermedia: Issues, Strategies, and Tactics in the Design of a Hypermedia Document System," Proc. of Conf. on Computer Supported Cooperative Work, Austin, December 1986, pp. 163-174.

[Greenberg 90]

Greenberg, S., "Sharing Views and Interactions with Single-user Applications," Proceedings of ACM COIS 90, pp. 227-237.

[Greif & Sarin 86]

Greif, I., and Sarin, S., "Data Sharing in Group Work," Proc. of Conf. on Computer Supported Cooperative Work, Austin, December 1986, pp. 175-183.

[Greif 88]

Greif, I. (ed.), Computer-Supported Cooperative Work , Morgan-Kaufmann Publishers, San Mateo, CA, 1988.

[Ishii 90]

Ishii, H., "TeamWorkStation: Towards a Seamless Shared Workspace," Proceedings of the Conference on Computer Supported Cooperative Work, Los Angeles, Oct. 1990, pp. 13-26.

[Johnson & Reichard 89]

Johnson, Eric F., Reichard, Kevin, "X windows applications Programming," MIS Press, Portland, Oregon, 1989.

[Kernighan & Pike 84]

Kernighan, B. W., and R. Pike, The UNIX Programming Environment Prentice-Hall, Englewood Cliffs, NJ, 1984.

[Lantz 86]

Lantz, K., "An Experiment in Integrating Multimedia Conferencing," Proceedings of the Conference on Computer-Supported Cooperative Work, Austin, TX.

[Lee 90]

Lee, J. J., "Xsketch: A Multi-user Sketching Tool for X11," Proceedings of ACM COIS 90, pp. 169-173.

[Leffler et al. 89]

Leffler, S. J., McKusick, M. K., Karels, M. J., Quarterman, J. S., The Design and Implementation of

the 4.3 BSD UNIX Operating System Addison-Wesley Publishing Company, Inc. 1989.

[Lewis & Hodges 88]

Lewis, B., and Hodges, J., "Shared Books: Collaborative Publication Management for an Office Information System," Proc. of Conf. on Office Information Systems, Palo Alto, March 1988, pp. 197-204.

[Newman-Wolfe et al. 91]

Newman-Wolfe, R., Ramirez, C., Pelimuhandiram, H., Wilson, D., and Webb, M., "The DCS Distributed Conferencing System" University of Florida CIS Dep't. TR-91-3, 1991.

[Ramirez & Pelimuhandiram 90]

Ramirez, C. and Pelimuhandiram, H., "Gossip: A Multi-User Talk Program Over the Internet," University of Florida CIS Dep't. TR-90-5, 1990.

[Ramirez 91]

Ramirez, Carmen L., "Networking Component of a Distributed Conferencing System," University of Florida CIS Dep't. Master's Thesis, 1991.

[Sarin & Greif 85]

Sarin, S. and Greif, I., "Computer-Based Real-Time Conferencing Systems," Computer, October 1985, pp. 33-45.

[Stefik et al. 86]

Stefik, M., Bobrow, D., Lanning S., and Tatar, D., "WYSIWIS Revisited: Early Experiences With Multi-User Interfaces," Proc. of Conf. on Computer Supported Cooperative Work, Austin, December 1986, pp. 276-290.

[Stefik et al. 87]

Stefik, M., Foster, G., Bobrow, D., Kahn, K., Lanning, S., and Suchman, L., "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings," Communications of the ACM, January 1987, pp. 32-47.

[Trigg et al. 86]

Trigg, R., Suchman, L., and Halasz, F., "Supporting Collaboration in NoteCards," Proc. of Conf. on Computer Supported Cooperative Work, Austin, December 1986, pp. 153-162.

[Trigg & Weiser 86]

Trigg, R., and Weiser, M., "TEXTNET: A Network-Based Approach to Text Handling," ACM Transactions on Office Information Systems, January 1986, pp. 1-23.

[Watabe, K., Sakata, S., Maeno, K., Fukuoka, H, and Ohmori, T.,

"Distributed Multiparty Desktop Conferencing System: MERMAID," Proceedings of the Conference on Computer Supported Cooperative Work, Los Angeles, Oct. 1990, pp. 27-38.

[Wilson 91]

Wilson, David L., "The Distributed Conferencing System User Manager," University of Florida CIS Dep't. Master's Thesis, 1991.

Richard E. (Nemo) Newman-Wolfe is an assistant professor of Computer and Information Science at the University of Florida, Gainesville. He attended Eckerd College (St. Petersberg) and earned his bachelor's degree in mathematics from New College (Sarasota) in 1981. His graduate studies at the University of Rochester (New York) concerned theoretical communication issues in parallel processing. He earned his Master's degree there in 1983 and completed his doctoral work in 1986. He has published over twenty technical papers describing his work in theory, networks, and systems. His current research activities include satellite communication using laser crosslinks, network traffic and performance modeling for MANs and CWANs, distributed collaboration, and computer support for urban and regional planning.

Carmen Lucia Ramirez earned her bachelor's degree in computer science in 1988 and her computer science Master's degree in 1990, both from the University of Florida. She contributed to the networking infrastructure for DCS. Her interests include operating systems and networking.

Harsha Pelimuhandiram is a Master's candidate in computer science at the University of Florida. He earned his bachelor's degree from the University of Texas (Austin) in 1988. He has worked on simulation of satellite networks in addition to many contributions to DCS, including the text editor.

Mario A. Montes earned his B.A. in computer science from Saint Mary's University (San Antonio) in 1981, and in 1985 his M.S. in management science from the University of Tennessee (Knoxville). From 1986 to 1987, he served as an Assistant Professor of Management at Georgia Southern College. Currently he is working toward his M.S. in computer engineering at the University of Florida. He is interested in personal computers, computer networks, and small business computer applications.

Michael L. Webb is a master of science student at the University of Florida in Gainesville, Florida. He received his bachelor's degree in computer science from Washington State University, Pullman, Washington, in May 1989. He was born in Tacoma, Washington, 1965. The Ensemble portion of DCS was written by Michael. Upon graduating from the University of Florida, Michael plans to be employed and working on a communications manager for a line of personal computers.

David Wilson received the B.S. degree in Computer and Information Sciences in 1987 from the University of Florida, and is currently working on an M.S., also at the University of Florida. He has worked at Intergraph Corp, and is presently employed at Encore Computer in Ft. Lauderdale, FL. He contributed the initial user manager and execute windows to DCS. His current interests include real-time Unix multiprocessing, and X clients and servers.

Software and Intellectual Property – Who Owns Your Work?

Moderator: *Rob Kolstad, Sun Microsystems*
Organizer: *Dan Geer, Digital Equipment Corporation*

Panelists will debate the basic issues of intellectual property, including legal, social, business, and technology aspects.

We'll explore whether the problems are well enough understood to craft solutions, or whether it must be left to the accretion of case law. Using specific proposals, we'll try to illustrate the consequences of the various world views. And, we'll consider how we might think about these issues as the very nature of digital information appears to be making national boundaries permeable, if not obsolete.

A Workstation-based Multi-media Environment For Broadcast Television

Keishi Kandori

Asahi Broadcasting Corporation
2-2-48,Oyodo-Minami Kita-ku Osaka 531 Japan
kk@media-lab.media.mit.edu

Abstract

This presentation will describe two applications which illustrate expanding workstation power in the real broadcasting world.One is a sophisticated application of virtual reality in which we simulated the environment of the green of the 18th hole in real time during a real professional golf tournament for a broadcast TV program.The other is professional baseball information management system for TV program production in which we incorporated distributed computing,mixed networking,database managing,and computer graphics.

1. Introduction

The multi-media computing environment for commercial Broadcast Television has recently become an important part of pre & post production. For these applications, it is necessary to incorporate information on paper, sound , motion video , and phone lines. The integration of all the above requires linked multi-media databases over high speed networks (FDDI, Ethernet, etc). Digital data is very convenient for computing, but real-time processing of digital audio and full-motion video is beyond the capabilities of any average computing environment . However, the power of computing has been increasing dramatically, and now multi-media work stations have emerged equipped with live video input, real time analog-to-digital conversion, compression, decompression, high-speed digital signal processing capabilities, and good graphics engines.

Here we will describe two examples where multi-media was applied to actual television broadcasting. One system was implemented for the television coverage of the "ABC Lark Cup Golf Tournament", of the Japanese PGA tours which ran from October 25 to 28 1990 at the ABC Golf Course near Osaka, Japan. In a golf tournament, the game is played at 18 different locations at the same time. That requires a good communication system and makes for a very interesting application for a workstation based multimedia environment. The broadcast center at the golf course monitors the game's data at every hole by live video cameras and personal computers connected via modem.

Before the tournament, we made a 3D geometric model of our golf course. This model was used to render the title animation of the TV program, and during the program we used it as the basic world model for a virtual camera looking at the golf tournament simulated with real-time computer graphics. At the real 18th green, we built a base for the live TV commentator from which he could see the whole 18th hole course, green, and the players. He also had a tracking ball which could control the location of the

virtual camera at the simulated 18th hole green. This was our brand of virtual reality as applied to golf.

Our other multimedia TV broadcasting example was a professional baseball application. This was a very wide area integrated network multimedia database incorporating both a main frame, workstations, and PC base computers, and utilizing many types of network protocols (X.25,TCP/IP,...). The multimedia database included text, graphics, and binary data. We accessed the database through the different on-line computers, and displayed the data on the computer screens and live TV screens in our service areas (all over Japan). The other interesting thing is that the Nikkan Data Sports Supply in Tokyo has started offering a video game using this on-line database (with a special game machine, anyone can play at home with this database).

2. Professional Golf Tournament

The goal of a TV golf tournament is to get viewers so involved that they feel that they are actually there at the course watching the star pro-golfers. One of the highlights is on the green of the 18th hole because that is where many dramatic and decisive events take place.

We used 3D computer graphics to model a complete 18 hole course and green. This model was prepared to start this project, about three month before the tournament began. As mentioned previously, this model was the basis for the virtual reality environment, and was also used for the TV title CG animation segments.

Our virtual reality system was composed of the following: (Figure 1,2) Figure 1 shows the equipment at the 18th green which comprised the golf ball position detector block. We positioned two live video cameras watching the ball on the real green. These video signals were overlaid on the PC screen, and the PC compared the signals in order to analyze and interpolate the 3D position of the golf ball. The results (3D coordination data) were transmitted by modems over a multi-wired communications cable.

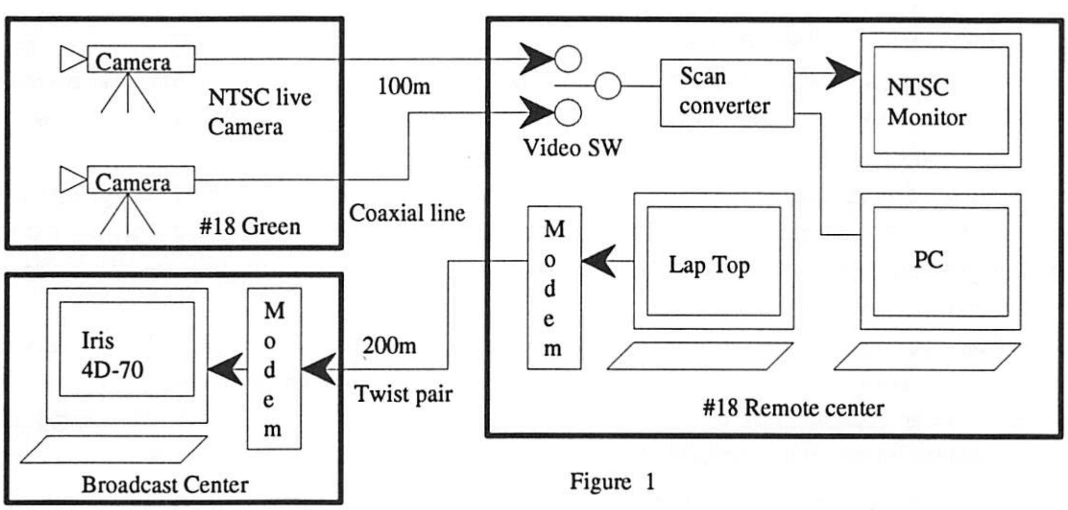


Figure 1

Figure 2 shows the broadcasting center at the golf course. An Iris 4D-70 managed the real-time CG animations. RGB output signals from the Iris were connected scan converted into Gen-locked center NTSC switcher signals. The broadcasting center at the golf course was linked by micro-wave connect to our main broadcasting center in Osaka (this was a traditional NTSC link), and broadcast all over Japan.

The creation of the golf course model entailed making triangular patches of the whole course, including the 18th hole and green. It took us about two months to make them. We rendered the database at

high quality for the title graphics and also used it in our real-time display system which had several modes. One is controlled by a commentator by means of a space ball and mouse. The space ball was used to control the display screen in order to rotate, and zoom in & out .. The mouse was used as the pointer device around the green. Others controls existed for the simulated viewpoints of all the real players, for simulated views to match those of the real TV cameras, and there were special programmed key switches to rotate the virtual green, rotate the whole virtual 18th hole course,etc..

Our whole system was robust enough to perform with great stability in the field during the entire tournament.

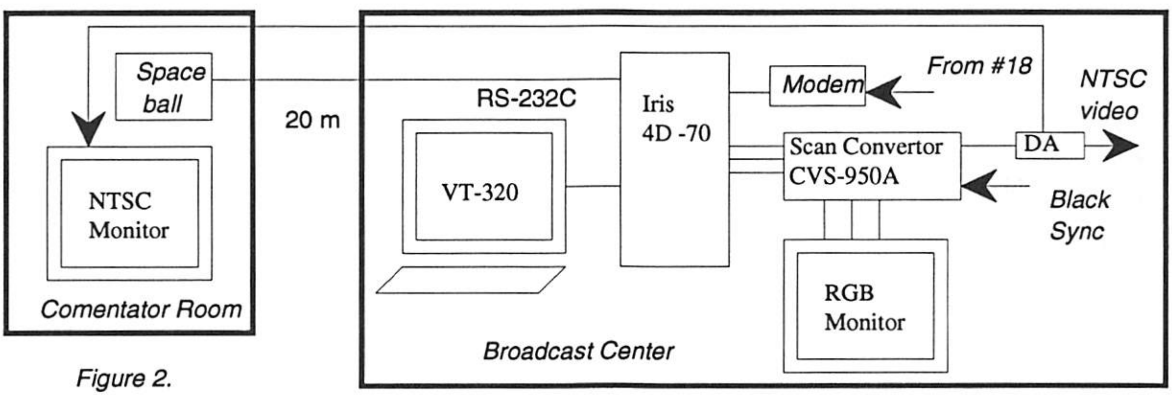


Figure 2.

3. Professional Baseball application

In Japan, there are twelve teams in two leagues. There are a maximum of six games in a single day all over Japan. Five years ago, our station started keeping baseball databases to assist the sports TV show announcers, show producers, and engineers. We used a microvax and our proprietary SDB database software. Our SDB software used the data collected from the previous day's games. At that time it was impossible to update the database with the results of the games on the same day. Three years ago we succeeded in making a database system for internal use only that could be updated in real-time by using a network of PCs.

Last year we improved the system by replacing the PC network with a network of multi-tasking multi-port unix workstation with a central database server . This now allows for increased capacity and real time online queries. This year we installed a commercial network database server by Sybase which uses the SQL language interface for the multimedia database.

Our current system is composed of three segments. Data acquisition from all over Japan, the internal multi-media data server, and visualization machines for TV & Radio Figure 3. shows whole diagram in Japan

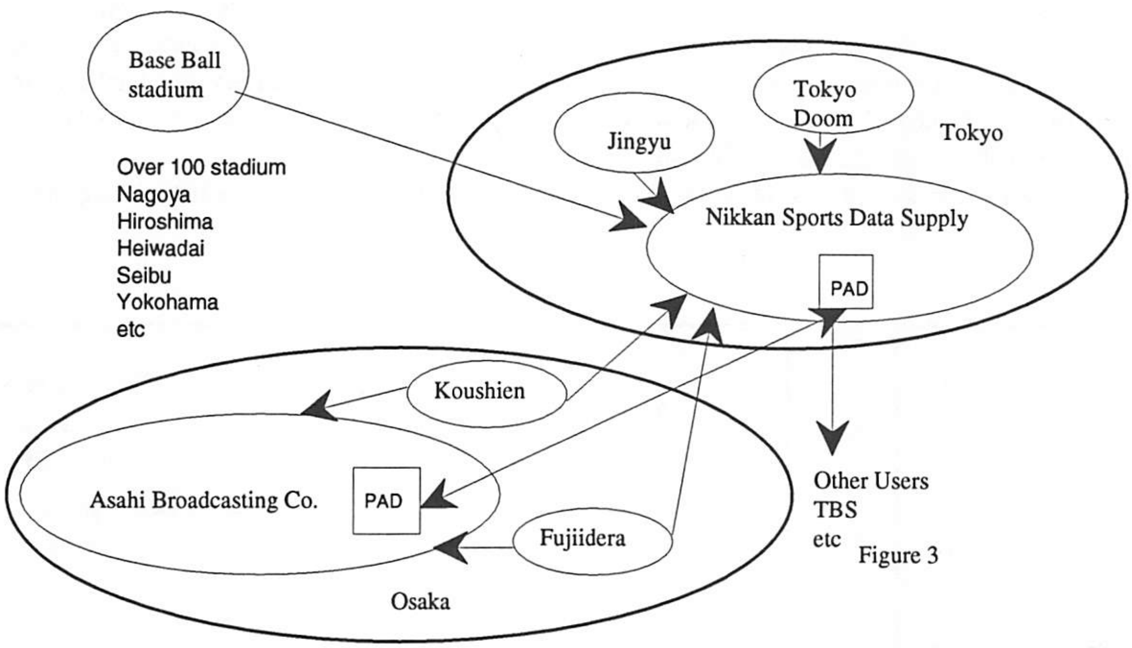


Figure 3

3.1 Data Acquisition

All of the online data is initially entered by PC-based laptops (IBM-5535) from baseball stadiums, and then sent to the main frame computer (IBM-4381) which is managed by NIKKAN SPORTS DATA SUPPLY in Tokyo. They control this data using their own database, and service many company through reserved data lines. One interesting application is a real time game service called "Pro Yakyu VAN" which is a cooperative project between Nikkan Sports Data Supply, Kyoudou-VAN, and Sega Enterprise using NTT dial Q2 lines.

3.2 Database and Network management System

We get the data through one X.25 line (between Tokyo and Osaka).

We have two systems. The older one which was named SDB is a Micro-Vax II, running VMS, and the Mumps environment. The SDB data comes in once a day after the all baseball games in Japan are finished. After all incoming the data from Tokyo is sent, the database is updated. Every morning SDB delivers the new database information. At the beginning of the season in April, SDB is initialized and a new baseball database is started, and at the end of baseball season in October, it is closed.

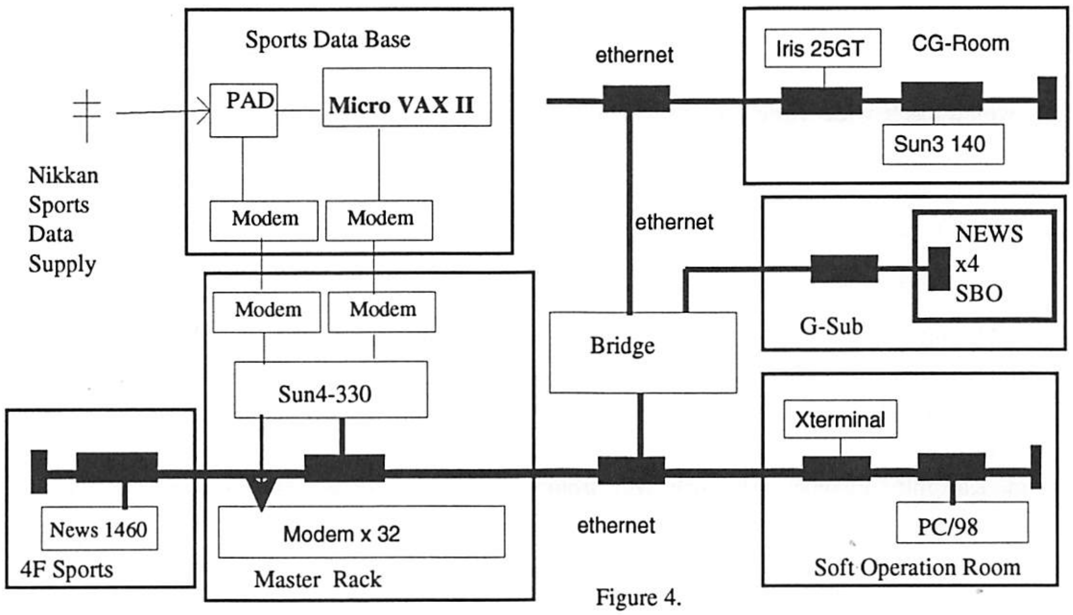


Figure 4.

The new one BNS system uses a Sun-4 330, and the Sybase SQL environment. This is run throughout all the seasons. The main units are connected by ethernet, and communication through this network is by transaction SQL. The baseball database is maintained by a Sony News work station located in the sports section. The basic database query services use modems and dumb text terminals.

3.3 Visualize system

The G Studio (one of our TV studios of which we have 6), is designed for live shows like golf, baseball, and football games. The computer resources in this room are four Sony News workstations linked by NIS (NFS). Two of the News have NTSC frame buffers which display on-line players' name, SBO, and TOKUTEN referenced in our SDB and BNS by modems & ethernet.

The C studio for general purpose TV, is used to produce the evening news show every day. In the news show we present current baseball information from this database. All of the data come from modems, and are displayed on PCs by our proprietary software.

4. Past and Future Direction

In 1986 we worked on making an Experimental Video Workstation which was a design project in computer-aided editing using an Audio/Video, Database and Computers, at the Film/Video Section of the MIT Media Laboratory. At that time we had only a small network, under 100 M storage device, and special user interfaces. The basic configuration was composed of and HP Bobcat, a PDP11 based UNIX machine, a U-matic VTR, laser discplayers, and an audio follow switcher. The project concept was to make a pictorial database for the computer-aided video editor for which we designed a virtual display & buttons on the HP starbase window.

In 1987 we built similar tools on a Sun-3, using MO disk storage for the NAB. This system had a very large capacity disk which could read & write A/V signals in real time. But the Sun windows interface was not powerful enough for our A/V information interfaces.

In 1988 we installed a Parallax board on the Sun, we re-designed the interface from Sun windows

to X11, and reported and demonstrated the results at IBEE Japan.

In the next few years we hope to see an international multimedia standard emerge so that we can upgrade our system to incorporate multimedia. In the meantime, we are investigating existing and planned multimedia formats which we can adopt in the near term. We are starting to design a multimedia data format which has a wide variety of font handling capabilities appropriate for NTSC and HDTV video production.

5. Acknowledgements

The author would like to acknowledge the following people from M.I.T. Media lab, Plus One Inc., Shumisho Electronics, and Nikkan Sports Data Supply for their help in making the Golf visualizing system, and the baseball database. Glorianna Davenport (MIT media lab), for her suggestion of the initial concept of experimental video workstation for NAB. Takeshi Sawai (Plus One Inc.), for 18th green special program. Katsuhiro Fukebaru (Sumisho Electronics Co.,LTD.), for his 3D visualizer on Iris. Katsuaki Higurashi (Nikkan Sports Data Supply), for on-line data. Takashi Ikeda (MI systems), for database server programmer. I also thank Mark Holzbach, (ShibaSoku Co.) for various consultations.

6. References

1. David R. Lampe "Athena Muse:Hypermedia in action"
2. Stewart Brand "The Media Lab"
3. Russell Mayo Sasnett "Reconfiguable Video"

Biography

Name : Keishi Kandori
Address : 2-Chome Oyodo-Minami Kita-ku Osaka City
Work Phone: 06-458-5321 (2766)
Home Phone: 06-835-1168
E-mail: kk@media-lab.media.mit.edu

Manager
Production Engineering Department Three
Asahi Broadcasting Corporation
May 1991

Visa president
Japanese alumni and friends of Media Laboratory
March 1991

President
NSUG(Nippon Sun User Group) of Osaka
February 1991

Plastic Editors for Multimedia Documents

Matthew E. Hodges, Digital Equipment Corporation
Russell M. Sasnett, GTE Laboratories Incorporated

Abstract. This paper describes our research with "plastic editors" in the Visual Computing Group of MIT's Project Athena. The idea of plastic editors is to break down the distinction between editor (program) and document (data), to enable the manufacture of editors which can be molded and shaped to meet many different needs. The *Athena Muse* authoring system [3, 4] allows us to experiment with the idea of building editing tools out of the components provided in the authoring system itself. In this paper we describe the importance of "mutable" editors in an application environment, the basic dimensions of plasticity in editors, and some examples of how we built and modified the Muse editing tools.

Interactive documents let the reader do more than just turn pages. Decades of research have gone into creating "electronic books," but the static organization and consumption patterns implied by that term have caused it to fall into disfavor. The current trend is away from prepackaged presentation modules, and towards more open information architectures providing separate storage, retrieval, and display mechanisms.

An author will always be an author; but the notion of a reader is in transition. An important threshold is crossed when the author decides to give the reader *generative* powers—creation, annotation, reorganization, customization.

For any of these changes, editing tools are neccessary—to produce new material, to modify the composition of a page, to change the ordering, or add cross-references from one part of the document to another. Usually, document preparation systems have *static* editors, compiled to perform a fixed set of tasks. They are designed to provide all the functions needed for a certain type of editing, and normally they are not meant to be changed.

When an author wants to give readers control, it is usually in a specialized context, where a particular type of modification is supported but readers do not have license to completely revise the entire work. General purpose editors are often inappropriate for these situations. Because they normally offer a broad set of functions, the control interfaces to these editors are often far too complex for casual readers. Also, it can be quite difficult to integrate the editor smoothly into the application—the interface may be inconsistent, the data may not transfer easily, and so on.

With the *Athena Muse* authoring system, we began experimenting with the notion of "plastic" editors—tools that can be molded to meet the specific requirements of an application.

The idea is to make the editor itself behave more like a document. In this way, application developers can use customized editors in their applications without having to modify and recompile some major part of an executable image.

Documents in Muse

The *Athena Muse* authoring system was designed to produce multimedia applications for the MIT faculty. The principal objectives for the system were to reduce the time and the level of skill needed to produce common forms of educational materials. The approach for achieving these goals was to build the system on a model of *document production* rather than one of *programming.*

Muse has been our vehicle for exploring multimedia authoring and document architecture for the past three years. We look upon it as a "thing to think with," a tool we have used to explore the requirements for non-specialist authoring by building real applications and interacting with real users in the MIT community.

Documents in Muse are not based on a *paper* metaphor. For example, a HyperCard stack can contain many types of information, but its structure and access mechanisms will always be bound to a paper metaphor: a stack of cards in a Rolodex. In a multimedia environment the limitations of a paper-based model quickly become a serious obstacle. This is because the dynamics inherent in electronic media are difficult to capture in fixed-size page units, and because multimedia systems are usually not designed solely to create paper output.

In contrast, Muse documents are built on a rich notion of multidimensional state. Each document unit, or *package*, is described by a set of *dimensions.* The complex state of a package is broken down into orthogonal "aspects" represented by each dimension. For example, one dimension could be the period of time over which the package is active, while another might indicate the language used for rendering text subtitles.

Muse Space

Each dimension defines a continuous domain of possible states between a minimum and maximum. Together the set of dimensions in a package describes an n-dimensional *space*, where the axes represent n degrees of freedom in the state of the package. Most Muse applications are created by placing content units at meaningful coordinates in the information space, and then providing the user with appropriate navigational controls to explore it.

The current positions of all the package dimensions can be concatenated to form a "package position" tuple, which completely describes one unique state of the package. Built-in *actions* can be invoked to operate on any package using these tuples—to set the package to a known state, to add an offset to a package's current position, and so on. Armed with a dozen or so actions, the new author is able to create a broad variety of multimedia applications with a minimal learning curve.

Perhaps the dimensional approach to problem specification in Muse is more natural to non-programmers, because it maps the abstract notion of *states* to the concrete notion of *locations in space.* Every unit of content in a Muse document is organized by its *position*

in a space of the author's making. Hence in *Athena Muse*, hypermedia links are made between two locations in information space (or two package states), rather than between two geographical positions in a document.

Most of the dynamic behavior of Muse documents is realized as a side-effect of changing a package's state via position tuples. Whenever a package changes position, all of its content *elements* which depend upon the changed dimensions are updated automatically. Since a position tuple can be encoded as a simple array of integers, it turns out to be a rather efficient means of update propagation. This is how multi-media synchronization is achieved in Muse documents.

Multimedia Synchronization

The foreign language materials produced by the Athena Language Learning Project [6] all required the display of text subtitles synchronized phrase-for-phrase with motion video. In Muse, we accomplish this by defining a package with a *frame* dimension and a *subtitles* dimension. Several button elements are placed into this package, which give the user control over the rate of a *timer* used to periodically fire one of the built-in positioning actions which updates the frame dimension. Similarly, a button is provided to turn the subtitles on or off by changing just the position of the subtitle dimension.

Muse provides many different types of displayable elements: motion video, single frames, digital images, scrolling text fields, buttons, and so on. Each type of element can have specialized behavior activated by the built-in dependency mechanisms.

For example, the playback rate of a video source automatically mirrors the rate of update on its *key dimension*. Likewise, the current frame displayed by the video element is computed from a specified range, which is mapped against some portion of the controlling dimension. The scrollbar element can be declared to *represent* a dimension, which causes it to become both an indicator and an interactive manipulator of that dimension's position. The text element can be directed to scroll itself to specified locations in a source document as a key dimension changes position. All of these elements can be used in various combinations to create special-purpose video viewing packages.

The above description should make it clear that synchronization in Muse goes beyond events triggered only by the advancement of "real" time. The dictionary defines synchronization as arranging for many things to happen at the same *time*; with Muse, we can arrange for many things to happen at the same *position* in a Muse state space.

In many ways, Muse has a much richer notion of synchronization than a simple event timeline. For example, all of the dimensions defining a space can be driven forwards or backwards with independent periodic timers. At any moment, the user may drag a scrollbar to pull one dimension back and forth at widely varying rates. In cases like these, it becomes impossible to pre-compute a single event timeline and wait patiently for intervals to lapse in order to effect synchronization.

The ability to describe events in *virtual* time is liberating because it generalizes time to be just another aspect of state. This is not to say that Muse can guarantee *hard real-time* performance, such as syncing digital audio and video streams with no more than 33 milliseconds of error; far from it. However, the ease with which multimedia synchronization can be expressed is more important at the application level than the implementation issues,

which should be encountered further down at the OS-interface level [1].

At the moment, Muse only handles synchronization inside a single process. GTE Laboratories plans to explore how independent Muse applications could be synchronized over a network using dimensions overlaid on Linda tuples [7].

Document or Editor?

The Muse document architecture was designed to accomodate dynamics as a fundamental attribute of information. For example, a simulation should be just another kind of document, not an out-board process controlled by back-door hooks. Muse makes it possible to create links directly into the state space of a simulation. These links might be used, for example, to provide additional information or to alert students to potential difficulties they may encounter.

Muse provides a set of basic building blocks that can be used to construct documents. These pre-defined objects are available in the *MusePackages* description language. A set of MusePackage objects are described, along with their attributes and interconnections, to form the bulk of an application. These objects can be extended or customized where neccessary by using a second part of the system, an object-oriented scripting language called *EventScript.* The scripts have the ability to execute compiled procedures from a library of "kernel" facilities.

The aim is to encompass as much of the content of the document in the pre-defined objects as possible. The degree to which this can be accomplished depends on how closely the application in question matches the requirements of the "application domain" Muse was designed to serve. For example, none of the target MIT applications required an interface to an external database. At GTE Laboratories, Muse was modified to support an interface to both an experimental distributed object manager (DOM) and an information retrieval engine (FAIRS) [8, 2].

The Muse document architecture was designed to reduce the difficulty of modifying components in-place. The more material is described as data, the easier it is to edit with Muse-crafted tools. One part of our research was to implement a set of editors using this approach. This would mean that the editor "documents" themselves could be handled as easily as the things they were designed to create and modify.

The challenge here is to identify the appropriate set of objects and methods that can "standardize" this activity—to minimize the customized behaviors, and where they are neccessary, to find conventional ways of handling them so they can be more easily transported from one application to another. The following sections describe some of the results and the issues encountered along the way.

Muse Editors

An "editor" is a set of tools for creating, deleting, modifying, combining or connecting information. Essentially, it consists of a set of functions operating on a known data format, where the data normally specifies displayable information such as video, text, graphics, or

audio.

Editors function at three levels: raw data definition; composition; and document interconnection. "Raw data" editors are those that produce simple forms of independent display material—pieces of text, graphics, video, audio, and so on. "Composition" editors combine these basic components into aggregates, handling the tasks of layout and media integration. "Document interconnection" editors control higher level organizational issues across sets of documents—cross-reference linkages, logical flow of control, query hierarchies, and so on.

We have built a number of interactive editors with Muse and explored some of the details of combining them into "workbench" combinations. The list includes editors at all three levels, although in some cases they are quite rudimentary:

1. Attribute editor: point-and-click window layout and appearance editing (font, color, etc.);

2. Subtitle editor: for synchronizing text subtitles with motion video sequences;

3. Pixmap editor: for clipping subregions of images and storing them in image sequence files;

4. Video editor: for defining video sequences and storing them in a local database;

5. Figure editor: for creating simple drawings;

6. Color Mixer: for defining rgb color combinations;

7. EventScript Browser: to inspect *EventScript* objects, variables, and methods;

8. Muse Builder: the best attempt to date at a Muse developer's workbench, combining many of the editors listed above along with allocation of windows, buttons, sliders, and support for dynamic editing of *EventScript* methods (Figure 1).

All of these editing tools are implemented as Muse documents. As noted earlier, Muse is not based on the usual "paper metaphor" common to most electronic document systems—the architecture is broad enough to encompass dynamic systems as a fundamental form of document.

The fact that these editors are implemented as Muse documents has at least one other important consequence—an editor can be included as a component of any other set of documents. An illustration of this point is provided by *Dans le Quartier St. Gervais*, a Muse application for teaching French developed at the Athena Language Learning Project [6]. The material is designed as a "discovery environment" where students can explore a database of imagery and interviews with people who live and work in an historic Paris neighborhood.

As part of the learning experience, the teachers who designed the material wanted students to be able to produce their own guidebooks for the neighborhood. The idea was that students should be able to collect sounds and images and assemble a personal "book" describing the highlights of what they discovered. This task requires a customized set of editors, designed especially to produce such a book quickly, easily, and in French (the aim, after all, is to teach French, not to turn students into multimedia producers).

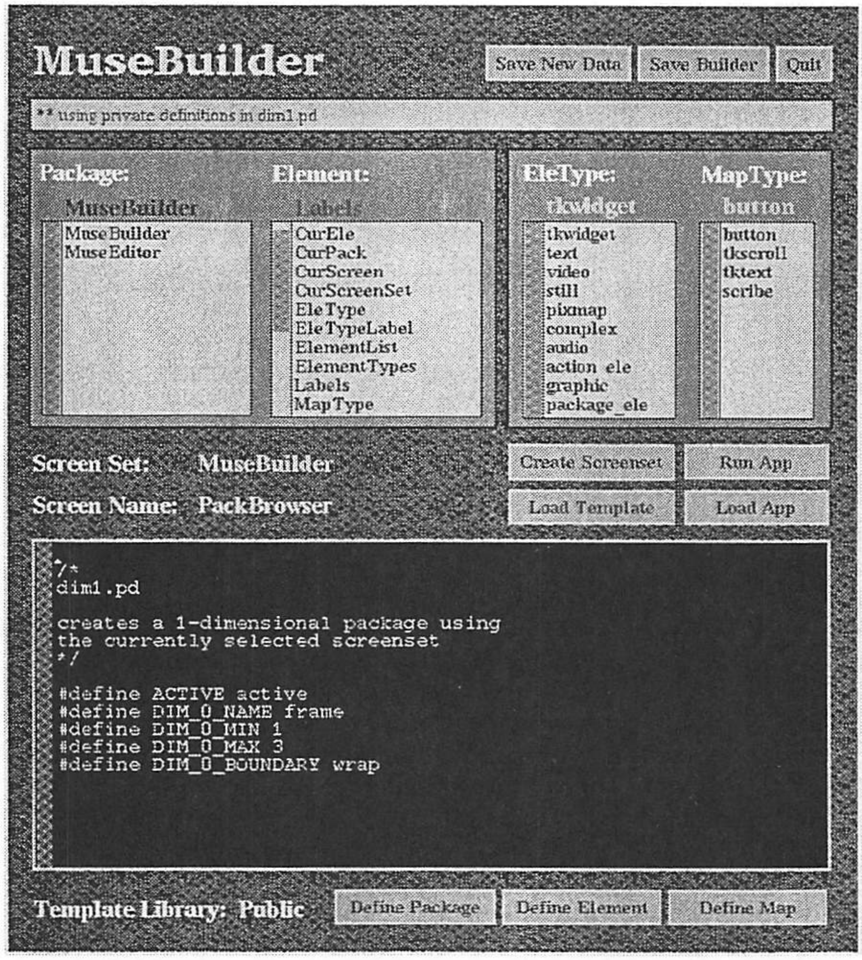


Figure 1: Primary control interface for the Muse Builder showing the selection mechanisms for the MusePackage objects and the template specification "scratchpad".

We made a small "book" editor in Muse, which gave students 100 blank pages and basic controls to take snapshots and add text. However, we realized they would need layout facilities and attribute editing to position and modify these contents. The solution was simply to include the entire Attribute Editor mentioned above as a content unit within the book. First, the editor was used to rearrange its own appearance; then some of its functions were removed, and a copy of the modifed editor was saved under a new name. Finally the "new" editor was simply plugged into the master document.

Dimensions of Plasticity

Editors can vary widely in their degree of *plasticity*—their ability to be molded and shaped to a new purpose. We have explored the following dimensions which characterize an editor's plasticity:

1. ease of appearance modification;
2. ease of behavior modification;
3. ease of integration with other applications;
4. ease of communication with other applications.

Mutable Appearance

Changing the appearance of an editor is the simplest form of modification. Usually it means changing the color, position, or size of user interface components. Any changes to the editor's appearance must be saved so they can be recalled at the next time of instantiation. Most X Toolkit applications support at least this degree of plasticity by using the X Resource Database to supply default values for a wide variety of widget attributes [9].

In Muse, any editor can have its appearance modified by the Attribute Editor. This editor can load any Muse application, modify its layout, color, text, or fonts, and generate a new copy of the Muse specification into a named file. Normally the "system copy" of an editor like the Muse Builder is write-protected, but copies can be made and modified to meet the requirements of other applications.

Mutable Behavior

Changing the appearance of an editor can be quite important, in terms of making it look and feel like part of some larger system. But it is usually neccessary to modify an editor's behavior as well. Behavior modification falls into two categories: *recombination* of pre-defined functionality, and *fabrication* of new functionality.

Recombination. Often the only behavior that needs changing in an editor is the set of functional mappings between events or conditions and the actions they trigger. For example, X Toolkit applications can change the behavior of a command button so that its callback routine is invoked on a *ButtonPress* event instead of a *ButtonRelease* event. This involves changing an attribute of the button called its Translation Table [9].

The central characteristic of recombinant behavior modification is that any functions or procedures used must already be available in the editor or base software system, so that no recompilation is required to access them. In essence, pre-defined functions are used to set the value of behavioral *attributes* in an editor's objects.

The Muse Video Editor provides a good example of how this works. Many applications require video "editing" of some kind; but the degree of complexity can vary greatly. Most

Muse video applications use independent "playback units" for viewing specific pieces of video content. This paradigm is so common that we frequently ask novice users to create a video control panel as their first exercise (see Figure 2).

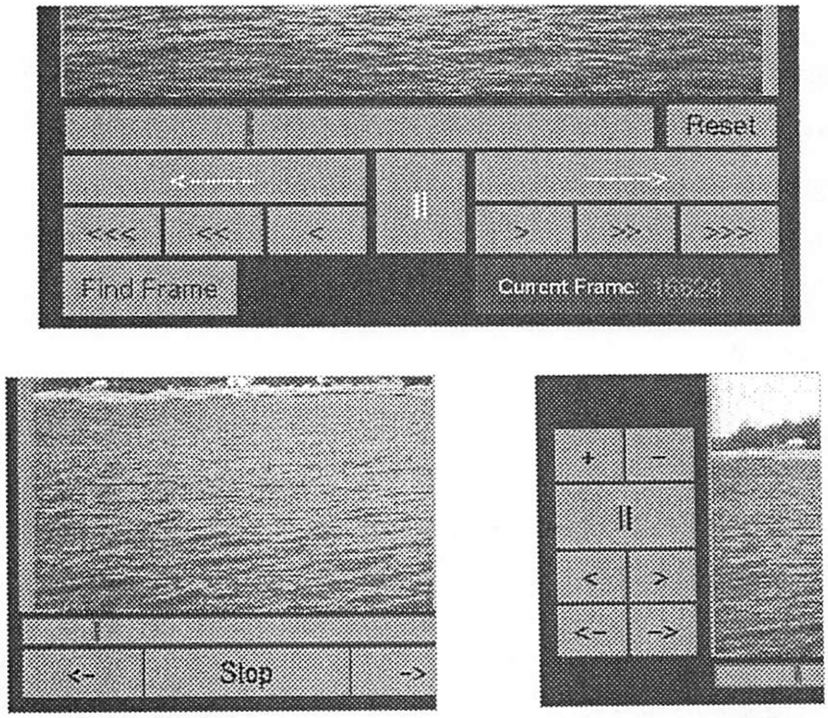


Figure 2: This illustration shows the control panels for three different video "playback units". The top figure shows the original control panel; the two lower figures were made by selecting subsets of the original functions (recombination) and then changing their appearance using the Muse Attribute Editor. The Video Tool in Figure 3 shows one more mutation of this control panel.

We have collected dozens of these video viewers over the past three years. All of them draw from the same list of built-in actions to control the display of the video material, but those actions can be combined in new ways with an endless variety of interface objects to trigger them. We refer to this as the "software deli" analogy: we can create editors made to order, just like sandwiches.

The actions used by a Muse video control panel operate on the "key dimension" of a video package. These actions are translated into media-specific functions that stop, start, or play a video device at various speeds in forward or reverse, or search to a particular frame. In fact, the video element could be removed from the control panel interface, and any other type of information element (such as a sequence of digital images) could be attached in its place. The editor would continue to work properly, because all Muse actions operate on generic package dimensions, and all elements know how to display themselves in response to dimension update events.

Some applications provide a full set of video controls, others only a few. The functions are all pre-defined, so modifying the editor is simply a matter of adding or deleting buttons and re-attaching library functions to them. These reconfigured control panels can usually be cut and pasted from one document to another without difficulty.

Fabrication. In contrast, a growing number of Muse applications need to load arbitrary video segments into a single application-wide viewer. These generic viewers are often just as simple as the control panels just described. But one of the more complicated viewers presents the user with a virtual "light table" interface, which can hold a large number of video "slides." The slides can be moved around the tabel, re-sized, and re-arranged to make conceptual groupings. None of this added-value functionality is built-in; it had to be fabricated by the document's author.

Figure 3: Control interface for the video editing tool. Parts of the interface have been adapted from a controller shown in Figure 2. Other functions have been fabricated to create and record video segments as a new type of entity.

Muse provides two avenues for fabricating functionality. The first choice is EventScript, the object-oriented scripting language in Muse. In the case of the Light Table editor, the author created a VideoSlide object as the "server" for a new data type (VideoSlide). A method was created to handle requests for creating new slides, with parameters such as a frame number and the identifier of the light table window. When the method is invoked, it creates a new VideoSlide object and places it on the light table. Additional methods are defined for VideoSlide to let it respond to mouse input for moving and resizing.

Clearly, the author of the Light Table editor is fabricating new functionality by procedural programming. This requires a correspondingly greater level of time and skill to perfect than simple Muse applications without the aid of EventScript. Instead of 16 functions to choose from, EventScript gives the author access to over 169 compiled functions in the Muse "kernel." This is an order of magnitude more complex than attaching a built-in action to a button, as described above. EventScript lies in the middle ground between the MusePackages language and C programming, in terms of power and ease of use. As a last resort, authors can write functions in C to be called from the EventScript interpreter.

Integration

Many times an application can be extended very easily by just embedding one or more stand-alone editors inside it. It should be possible to uproot an editor, transplant it from one context to another, and graft it onto a previously independent application with a minimum of effort.

Well-designed editors should reduce or eliminate their external dependencies, except for initialization messages which "bind" them into an application environment. The goal is to be able to embed editors into ordinary documents, as in the student-generated guide books described above in the French application.

Communication

Some editors and the functions they provide are so pervasive or generic that it doesn't make sense to embed them in many different applications or documents. A single editor can function as a "server" to any number of client applications.

These editors can work in several different communications modes. Some will just broadcast universal selections, such as a color or font name. Others will pipe continuous streams of input or output to clients. But the most powerful editors will in essence become fully manipulable "children" of their clients, with data passed back and forth over a messaging protocol.

All of these editors need a way to rendezvous with each other, and perform inter-process communication. Since by defintion the editors have an X Windows user interface, it seems logical to use the Xlib API to locate processes at the X server, and to pass messages between them.

We experimented with two independent editor applications, one a color palette and the other a font selector, which communicated with the Muse Attribute Editor using the X Selection mechanism [5]. This strategy leads to a very "loose" integration of editors through communications channels, in contrast with the embedded integration approach outlined in the previous section. Completely different font and color selection programs can be written to provide the basic functionality, but the application(s) on the receiving end of the X Selection will never know the difference.

Summary

The idea of plastic editors becomes important when authors want to give readers control. This is an area that is not well addressed with existing approaches to document preparation. It is something that we are finding increasingly important in our work with programming environments for non-specialists.

The challenge we see is to increase the flexibility of editing tools without substantially increasing the burden of programming them. With Muse, we found that we could produce a variety of the higher-level editors and customize them to meet the needs of particular applications. This was largely because of the base architecture of the documents that was rich enough to support the basic level of interaction needed to implement such an editor.

However, in many cases, these editors required significant amounts of procedural code, in EventScript and occasionally in C. This unfortunately places some forms of customized editors out of the reach of most authors. The aim of our future research in this area is to simplify the process of document production while increasing the plasticity of the editors involved.

Matthew Hodges is a senior software engineer with Digital Equipment Corporation and visiting scientist at MIT Project Athena. He is acting as software manager for Athena's Visual Computing Group and is co-architect of the *Athena Muse* multimedia authoring system.

Russell Sasnett is a senior member of the technical staff at GTE Laboratories. He is co-architect of the *Athena Muse* authoring system, and chief engineer of the plastic editors described in this paper.

References

[1] D.P. Anderson, S. Tzou, R. Wahbe, R. Govindan, and M. Andrews. Support for continuous media in the DASH system. *Proceedings of the 10th International Conference on Distributed Computing Systems*, May 1990.

[2] A. Chow. Towards a Friendly Adaptable Information Retrieval System. *Proceedings of RIAO '88*, March 1988.

[3] Matthew E. Hodges, Russell M. Sasnett, and Mark S. Ackerman. A construction set for multimedia applications. *IEEE Software*, January 1989.

[4] Matthew E. Hodges, Russell M. Sasnett, and V. Judson Harward. Musings on multimedia. *UNIX Review*, February 1990.

[5] Oliver Jones. *Introduction to the X Window System.* Prentice Hall, 1989.

[6] David R. Lampe. Athena Muse: Hypermedia in action. *The MIT Report*, February 1988.

[7] Wm. Leler. Linda meets UNIX. *IEEE Computer*, February 1990.

[8] F. Manola, M.F. Hornick, and A.P. Buchmann. Object data model facilities for multimedia data types. *GTE Laboratories Technical Memorandum, TM-0332-11-90-165*, December 1990.

[9] Douglas Young. *Programming and Applications with Xt.* Prentice Hall, 1989.

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *;login:*; produces a quarterly technical journal, *Computing Systems*; co-publishes books with The MIT Press; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

Computing Systems, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and in addition, the Association will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX users and system maintainers who convened periodically to discuss problems and exchange ideas concerning UNIX. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the fees paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association
Suite 215
2560 Ninth Street
Berkeley, CA 94710
Telephone: 415/528-8649
Email: office@usenix.org
Fax: 415/548-5738

USENIX SUPPORTING MEMBERS

Aerospace Corporation
AT&T Information Systems
Digital Equipment Corporation
Frame Technology, Inc.
Quality Micro Systems
Matsushita Graphic Communication Systems, Inc.
mt Xinu
Open Software Foundation
Sun Microsystems, Inc.
Sybase, Inc.
UUNET Technologies, Inc.